FINAL REPORT

Contract N00014-85-K-0159

ON-LINE ARITHMETIC ALGORITHMS AND STRUCTURES FOR VLSI

December 15, 1984 - December 14, 1987

Office of Naval Research
Contract No. N00014-83-K-0493

Principal Investigator

Professor Miloš D. Ercegovac

Faculty Associate:

Professor Tomas Lang

UCLA Computer Science Department
University of California, Los Angeles
Los Angeles, California 90024
(213) 825-2660

November 1988

Table of Contents

1. Summary of the Project Objectives 3

2. Summary of Contributions 3

A. On-Line Algorithms and Designs 3
B. Design Methods for Matrix Computation Arrays 5

3. Graduate Students 7

4. Publications Resulting from This Project 7

Appendices: Selected Publications

1. D. Tullsen and M.D. Ercegovac, *Design and Implementation of An On-Line Algorithm*, SPIE Conference on Advanced Architectures for Signal Processing, San Diego, August 21, 1986.

2. M.D. Ercegovac and T. Lang, "On-Line Schemes for Computing Rotation Angles for SVDs", Proc. SPIE Conference on Real-Time Signal Processing, San Diego, August 1987.

3. M.D. Ercegovac, T.Lang, and J.G. Nash, "An Area-Time Efficient Binary Divider", Proc. ICCD '87 Conference, New York, 1987.

4. M.D. Ercegovac and T. Lang, "On-Line Scheme for Computing Rotation Factors", Journal of Parallel and Distributed Computing, Vol.5, June 1988, pp. 209-227.

5. J. Moreno, "A Proposal for the Systematic Design of Arrays for Matrix Computations," Report No. CSD-870019, Computer Science Department, UCLA, May 1987.

6. Moreno, J.H. and T. Lang, "On Partitioning the Faddeev Algorithm", Proc. Intl. Conf. on Systolic Arrays, May 1988, San Diego.

1. SUMMARY OF THE PROJECT OBJECTIVES

The research and development problem we have investigated in this project is the VLSI implementation of fast and highly parallel algorithms based on on-line arithmetic. The on-line approach is characterized by simple interconnection requirements and digit-level pipelining suitable for highly concurrent special-purpose VLSI designs. The objective of the project was to evaluate the feasibility and efficiency of on-line approach in NMOS and CMOS VLSI implementations and study its use in signal processing and matrix computations. The project involved the following tasks: (A) development or selection of on-line algorithms suitable for VLSI implementation, (B) bit-level design and simulation, (C) NMOS and CMOS circuit-level design and simulation, (D) VLSI chip implementation, (E) performance measurements and evaluation, (F) development of design methodologies for special-purpose arithmetic-intensive architectures, and (G) applications of on-line/redundant algorithms to signal processing and matrix computations.

2. SUMMARY OF CONTRIBUTIONS

Part A: On-Line Arithmetic Algorithms and Designs

In the area of on-line and redundant arithmetic algorithms and VLSI designs, several contributions have been made:

2.1 VLSI Design of an On-Line Module [1, R1]

An NMOS circuit design of the operator $Y = AX + B$ in on-line arithmetic has been completed, the chip was layed out, extensively simulated, and fabricated via the ISI-MOSIS facility. The design has several novel features which make the critical path (the cycle time) very short: about 3 gate delays. The design is very regular with almost fully overlapped interconnect and active device area. The design is modular and can be used to obtain different operand precision without design changes.

2.2 On-the-Fly Conversion of Redundant Representation [4]

An algorithm to convert redundant number representations into conventional representations has been developed. The algorithm is performed concurrently with the digit-by-digit generation of redundant forms by schemes such as SRT division. It has a step delay roughly equivalent to the delay of a carry-save adder and simple implementation. The conversion scheme is applicable in arithmetic algorithms such as nonrestoring division, square root, and on-line operations in which redundantly represented results are generated in a digit-by-digit manner, from most significant to least significant.

2.3 On-Line Scheme for Computing Rotation Factors [5, 13]

A VLSI chip implementing the integrated radix-2 floating-point on-line algorithm for computing rotation factors for matrix transformations has been developed. The inputs are in parallel form, conventional Sign and Magnitude, floating-point representation. The outputs can be used in on-line signed-digit or in parallel form. The exponents are computed using conventional arithmetic while the significands are processed using on-line algorithms. The conventional result is obtained by using an on-the-fly conversion scheme. The rotation factors are computed in $10+n$ clock cycles for n-bit significands. The clock period is kept small by the use of carry-save adder schemes. A CMOS design and the layout phase have been completed and circuit/functional simulations are in progress. The design is expected to be sent to MOSIS for implementation in October 1988.

2.4 On-Line CORDIC Algorithm for Sine/Cosine Computation [10]

An on-line CORDIC algorithm for computing the sine and the cosine of a given angle has been developed. Its key features are: (i) use of redundant digit set for angle decomposition which allows carry-save addition in the angle recurrence, (ii) an on-line implementation of the recurrences for x and y, replacing variable shifters by area-efficient shift-register delays, (iii) on-the-fly conversion of results into the conventional representation [4], and (iv) an overlapped computation of the correction factor. The overall delay is about $3n$ clock periods with an expected speedup of 2 with respect to a conventional CORDIC implementation.

2.5 Redundant and On-Line CORDIC for Givens Rotations and SVD [R6, 12,14]

Several modifications to the CORDIC method have been introduced in order to improve speed and efficiency of its implementation when it is used for calculating angle and rotation for Givens' method. The main contributions are: (i) the introduction of redundant (carry-free) addition to replace time-consuming conventional additions; (ii) the use of on-line arithmetic to reduce the communication bandwidth, maximize the overlap between successive operations, and replace area-expensive shifters by delays; (iii) the use of angles in decomposed forms to eliminate angle accumulation recurrences. These modifications contribute to a speedup of about 4.5 with respect to standard CORDIC. Some considerations are given with respect to the complexity of implementation; however, a more detailed analysis would require actual VLSI implementation.

Two floating-point radix-2 schemes using on-line arithmetic for implementing the direct two-angle method for SVDs have been developed. The first scheme is an on-line variant of the cosine/sine approach and is the fastest of the schemes considered: it performs the 2×2 SVD step in about $2n$ clock cycles. However, it requires a relatively large number of modules; this number is reduced when some modules are reused, resulting in a time of $3n$ clock cycles. The number of modules of this on-line version is still larger than that of the conventional one, but this is compensated by the smaller number of bit-slices per module and by the digit-serial communication among modules. The corresponding speed-up ratios are of 5 and 3 with respect to a conventional arithmetic implementation. The second scheme uses an on-line CORDIC

approach and performs the 2×2 SVD in about $7n$ clock cycles and is advantageous because it is more time-area efficient. It results in a speed-up of about 2.5 with respect to the conventional CORDIC implementation.

An implementation of the diagonal and off-diagonal processors for an array performing the singular value decomposition (SVD) was developed. The implementation uses a modification of the CORDIC module that utilizes carry-save addition instead of carry-propagate addition, resulting in a significant improvement in speed of about 4 with respect to the conventional CORDIC.

2.6 SRT Divider with On-the-Fly Conversion [8,11]

An NMOS circuit design and implementation of a 32-bit fixed-point divider with the following features has been completed in cooperation with Hughes Research Laboratories, Malibu, California. The recurrence uses a 3-to-2 carry-save adder to form partial remainders. The quotient digits in the set {-1,0,1} are selected using the SRT method on the basis of 4-bit estimate of the scaled partial remainder and independently of the divisor. The conventional 2's complement form of the quotient with digits in the set {0,1} is obtained concurrently with the recurrence steps using our on-the-fly conversion algorithm. The chip has been implemented in a conservative technology (3 micron NMOS) demonstrating a very regular and dense design, high speed operation (32 MHz clock), and low power dissipation of 7 mW per bit. The division implementation is compatible in speed and area to a multiplier which simplifies design of a systolic processor chip for linear algebra applications and optimizes its performance.

2.7 Radix-4 On-Line Division [4]

A radix-4 floating-point division algorithm has been developed. In order to simplify the quitient-digit selection function, the divisor is transformed into a range such that the quotient digits are computed as a function of the scaled partial remainder only.

Part B: Design Methods for Matrix Computation Arrays

In the area of design methods for special purpose arrays, the following are some of the obtained results:

2.8 Design of Matrix Computation Arrays [2,3,9, R3, R4]

Our current research in the design aspects for special purpose arrays is oriented towards the development of a design methodology for matrix algorithms, with the capability to handle and relate features of the algorithm and the implementation in a unified manner. This methodology provides mechanisms to deal with issues such as data broadcasting, data synchronization, interconnection structure, I/O bandwidth, number of PEs, throughput, delay, and utilization of PEs. We have proposed a methodology based on the dependence graph of algorithms. Starting from a fully-parallel graph, in which nodes represent the operations and

edges correspond to data communications, we apply transformations to the graph to incorporate the issues listed above. The specific transformations depend on the particular parameter of interest at a given time.

2.9 Partitioned Implementation of Faddeev Algorithm [16]

We have considered the application of a graph-based methodology to derive partitioned implementations for the Faddeev algorithm. We have obtained linear and two-dimensional arrays for such algorithm, and we have compared these structures to others previously proposed. We have shown that the two-dimensional array derived here is more efficient and has less overhead than those other schemes. Moreover, we have shown that linear and two-dimensional arrays exhibit the same I/O bandwidth from the host, and utilization and throughput of both structures tend to the same values. We have concluded that, since performance measures of both arrays are identical, a linear array is better than two-dimensional one because it is simpler to implement and is more suitable to incorporate fault-tolerant capabilities.

2.10 Arrays for Partitioned Matrix Algorithms [17]

We have addressed tradeoffs between local storage and cell communication bandwidth in the design of arrays for matrix computations. We have presented a graph-based partitioning method to map matrix algorithms to different types of arrays. With the method, it is possible to trade between local storage in a cell and cell communication bandwidth, thus reducing the communication bottleneck that characterizes systolic cells. Moreover, the method facilitates exploiting pipelining within cells. The partitioning method also allows evaluating tradeoffs between linear and two-dimensional arrays. With our method, a designer can determine the cell type required for an implementation based on the maximum values possible for cell communication bandwidth and functional unit computation rate, parameters that depend in the technology used.

2.11 Partitioning of Matrix Algorithms for Systolic Arrays [18]

We have proposed a technique to partition algorithms for execution in arrays, based on transformations to the dependency graphs of algorithms. We described the application of such technique to the computation of transitive closure of a directed graph. This technique is suitable for a class of important matrix algorithms, produces implementations with maximal utilization of cells and no overhead due to partitioning, and allows evaluating trade-offs between linear and two-dimensional structures. We derived linear and two-dimensional arrays for partitioned computation of transitive closure. In the process we have obtained a dependence graph which is suitable for implementation of a fixed-size array for transitive closure, with better characteristics than structures previously proposed for this algorithm.

3. Graduate Students

Several graduate students have been involved in the research on this project:

Paul Tu, Ph.D. candidate, degree expected Winter 1989.
Thesis title:"On-Line Algorithms in Signal Processing Applications: Implementation of SVD"

Jaime Moreno, Ph.D. candidate, degree expected Winter 1989.
Thesis title: "A Proposal for the Systematic Design of Arrays for Matrix Computations",

Dean M. Tullsen, M.S. Degree in Computer Science received June 1986.
Thesis title: "A Very Large Scale Implementation of an On Line Arithmetic Unit"

Steve Faris, M.S. candidate, degree expected Fall 1988.
Thesis title: "A CMOS VLSI Design and Implementation of An On-Line Rotation Algorithm"

Li-Ken Tang, Ph.D. Candidate, worked on the project during 1986.

Charles Tong, Ph.D. Candidate, worked on the project during 1987.

Acknowledgements

The contributions by Steve Faris, Jaime Moreno, Paul Tu, and Dean Tullsen, who carried out parts of this research and development, are gratefully acknowledged. June Myers and Marilyn Kell provided efficient and friendly administrative and secretarial help.

4. PUBLICATIONS RESULTING FROM THIS PROJECT

Journal and Conference Papers

1. D.M. Tullsen and M.D. Ercegovac, *Design and Implementation of An On-Line Algorithm*, SPIE Conference on Advanced Architectures for Signal Processing, San Diego, August 21, 1986.

2. J. Moreno and T. Lang, "Multilevel Pipelined Processor for the Single Value Decomposition", SPIE Conference on Advanced Architectures for Signal Processing, San Diego, August 21, 1986.

3. J. Moreno and T. Lang, "Replication and Pipelining in Multi-instance Algorithms", International Conference on Parallel Processing, August 24, 1986.

4. Ercegovac, M.D. and Lang, T., "On-the-Fly Conversion of Redundant into Conventional Representations", IEEE Transactions on Computers, C-36, No.7, July 1987, pp.895-897.

5. Ercegovac, M.D. and T. Lang, "On-line Scheme for Computing Rotation Factors", Proc. 8th IEEE Symposium on Computer Arithmetic, 1987.

6. Tu, P. and M.D. Ercegovac, "A Radix-4 On-Line Division Algorithm", 8th IEEE Symposium on Computer Arithmetic, 1987.

7. Ercegovac, M.D. and T. Lang, "On-Line Schemes for Computing Rotation Angles for SVDs", Proc. SPIE Conference on Real-Time Signal Processing, San Diego, August 1987.

8. Ercegovac, M.D., T. Lang, J.G. Nash, "An Area-Time Efficient Binary Divider", Proc. ICCD '87 Conference, New York, 1987.

9. Moreno, J. and T. Lang, ''Design of Special-Purpose Arrays for Matrix Computations: Preliminary Results,'' SPIE Real-Time Signal Processing X, 1987.

10. Ercegovac, M.D. and T. Lang, "Fast Cosine/Sine Algorithm Using On-Line Cordic", IEEE Asilomar Conference on Signals, Systems, and Computers, 1987.

11. Nash, J.G., L.W. Chow, M.D. Ercegovac, and T. Lang, "Implementation of a Serial/Parallel Multiplier and Divider on a Systolic Chip", IEEE Asilomar Conference on Signals, Systems, and Computers, 1987.

12. Ercegovac, M.D. and T. Lang, "Implementation of Fast Angle Calculation and Rotation Using On-Line Cordic", Proc. 1988 IEEE International Symposium on Circuits and Systems, Helsinki, Finland, June 1988.

13. Ercegovac, M.D. and T. Lang, "On-Line Scheme for Computing Rotation Factors", Journal of Parallel and Distributed Computing, Vol.5, June 1988, pp. 209-227.

14. Ercegovac, M.D. and T. Lang, "Implementation of an SVD Processor Using Redundant CORDIC", Proc. SPIE Conference on Real-Time Signal Processing, San Diego, August 1988.

15. Ercegovac, M.D., T. Lang, and R. Modiri, "Implementation of Fast Radix-4 Division with Operands Scaling", Proc. IEEE International Conference on Computer Design: VLSI in Computers and Processors, New York, October 1988.

16. Moreno, J.H. and T. Lang, "On Partitioning the Faddeev Algorithm", Proc. Intl. Conf. on Systolic Arrays, May 1988, San Diego.

17. Moreno, J.H. and T. Lang, "Arrays for Partitioned Matrix Algorithms: Tradeoffs Between Cell Storage and Cell Bandwidth", Proc. SPIE Conference on Real-Time Signal-Processing, 1988, San Diego.

18. Moreno, J.H. and T. Lang, "Graph-based Partitioning of Matrix Algorithms for Systolic Arrays: Application to Transitive Closure", Intl. Conf. on Parallel Processing, 1988.

Technical Reports:

R1. Tullsen, D.M., "A Very Large Scale Integration Implementation of An On Line Arithmetic Unit", MS Thesis, UCLA Computer Science Department, CSD-860094, June 1986.

R2. Ercegovac, M.D. and T. Lang, "Simple Radix-4 Division with Divisor Scaling", Report No. CSD-870015, March 1987.

R3. Moreno, J., "A Proposal for the Systematic Design of Arrays for Matrix Computations," Report No. CSD-870019, Computer Science Department, UCLA, May 1987.

R4. Moreno, J., and T. Lang, "Removing Algorithm Irregularities in the Design of Arrays for Matrix Computations," Report No. CSD-870040, Computer Science Department, UCLA, August 1987.

R5. Ercegovac, M.D. and T. Lang, "On-Line Schemes for Computing Rotation Angles for SVDs", Report No. CSD-870043, August 1987.

R6. Ercegovac, M.D. and T. Lang, "Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD", Report No. CSD-870046, September 1987.

R7. Ercegovac, M.D. and T. Lang, "Fast Multiplication Without Carry Propagate Addition", September 1987, CSD-870047.

R8. Ercegovac, M.D. and T. Lang, "Radix-4 Division with Scaling and Quotient Digit Prediction", July 1988, CSD-880049.

R9. Moreno, J.H. and T. Lang, "Reducing the Number of Cells in Arrays for Matrix Computations", March 1988, CSD-880014.

Presentations:

M.D. Ercegovac, "VLSI-Oriented Algorithms and their Specification", Computer Science Department Seminar, University of California, Santa Barbara, March 7, 1986 (invited).

M.D. Ercegovac, "On-line Scheme for Computing Rotation Factors", 8th IEEE Symposium on Computer Arithmetic, Como, Italy, May 19-21, 1987.

M.D. Ercegovac, "A Radix-4 On-Line Division Algorithm", 8th IEEE Symposium on Computer Arithmetic, Como, Italy, May 19-21, 1987.

T. Lang, "On-Line Schemes for Computing Rotation Angles for SVDs", SPIE Conference on Real-Time Signal Processing, San Diego, August 1987.

J. Moreno, ''Removing Algorithm Irregularities in the Design of Arrays for Matrix Computations,'' SPIE Conference on Real-Time Signal Processing, San Diego, August 1987.

M.D. Ercegovac, "Fast Arithmetic for VLSI Architectures", University of Southern California, School of Engineering, Department of Electrical Engineering-Systems, January 22, 1988 (invited).

T. Lang, "Fast Cosine/Sine Algorithm Using On-Line Cordic", IEEE Asilomar Conference on Signals, Systems, and Computers, November 1987.

J.G. Nash, "Implementation of a Serial/Parallel Multiplier and Divider on a Systolic Chip", IEEE Asilomar Conference on Signals, Systems, and Computers, November 1987.

T. Lang, "Implementation of an SVD Processor Using Redundant CORDIC", SPIE Conference on Real-Time Signal Processing, San Diego, August 1988.

T. Lang, "Implementation of Fast Radix-4 Division with Operands Scaling", IEEE International Conference on Computer Design: VLSI in Computers and Processors, New York, October 1988.

J.H. Moreno, "Design of Special-Purpose Arrays for Matrix Computations: Preliminary Results", SPIE Conf. on Real-Time Signal-Processing, San Diego, 1987.

J.H. Moreno, "On Partitioning the Faddeev Algorithm", Intl. Conf. on Systolic Arrays, May 1988, San Diego.

J.H. Moreno, "Arrays for Partitioned Matrix Algorithms: Tradeoffs Between Cell Storage and Cell Bandwidth", SPIE Conference on Real-Time Signal-Processing, 1988, San Diego.

J.H. Moreno, "Graph-based Partitioning of Matrix Algorithms for Systolic Arrays: Application to Transitive Closure", Intl. Conf. on Parallel Processing, 1988.

Reprinted from SPIE Vol 696—Real Time Signal Processing IX

DESIGN AND VLSI IMPLEMENTATION OF AN ON-LINE ALGORITHM

Dean M. Tullsen and Miloš D. Ercegovac
UCLA Computer Science Department
University of California, Los Angeles

Abstract

We present a design and its VLSI implementation of a radix-2 on-line algorithm for the basic function $Y = AX + B$ in NMOS technology and discuss its area/time characteristics. The design uses internal pipelining to achieve a short step time of about three gate delays. The on-line delay is 5. The implementation is modular using a 150-transistor bit-slices. We also illustrate the use of the module in implementing a root solver for a polynomial equation.

1. Introduction

In on-line computations the operands, as well as the results, flow through arithmetic units in a digit-by-digit manner starting with the most significant digit [ERCE84]. These algorithms are such that in order to generate the j-th digit of the result $(j+\delta)$ digits of the corresponding operands are required. The on-line delay δ is usually a small integer. Successive operations execute in an overlapped manner as soon as δ input digits are available. In the conventional digit-serial arithmetic, in general, all digits must be known before a successive operation begins. Since digit-serial arithmetic reduces the interconnection bandwidth, on-line arithmetic is attractive in high-speed multi-module structures for parallel and pipelined computations where full precision bandwidth between the modules is not desirable or feasible.

In Section 2 we discuss the on-line algorithm for $AX+B$. The design of this algorithm is presented in Section 3 with an emphasis on internal pipelining of the recurrence. The area/time characteristics of its implementation in 4μ NMOS technology are given in Section 4. In Section 5 we illustrate the use of the module in implementing a root solver for a polynomial equation. Further details on the design and implementation are in [TULL86a].

2. The Algorithm

The arithmetic unit implements the arithmetic expression AX+B with A and X on-line, most significant digit first, in a radix-two signed-digit format. B is assumed to be available off-line in two's complement form. Minor modifications to the algorithm are required to accept B in on-line form. Outputs are produced on-line in signed-digit form. The derivation of the algorithm follows [ERCE75, TRIV77].

The operands A and X are fractions represented in on-line form as

$$X_j = \sum_{i=0}^{j} x_i 2^{-i} = X_{j-1} + x_j 2^{-j} \quad x_i \in \{-1,0,1\} \tag{2.1}$$

$$A_j = \sum_{i=0}^{j} a_i 2^{-i} = A_{j-1} + a_j 2^{-j} \quad a_i \in \{-1,0,1\}$$

The product at the jth step is:

$$\begin{aligned} X_j A_j + B &= X_{j-1}A_{j-1} + X_{j-1}a_j 2^{-j} + A_{j-1}x_j 2^{-j} \\ &\quad + x_j a_j 2^{-2j} + B \\ &= X_{j-1}A_{j-1} + (X_j a_j + A_{j-1}x_j)2^{-j} + B \end{aligned} \tag{2.2}$$

Let P_j be the scaled partial product at step j:

$$P_j = X_j A_j 2^j + B \tag{2.3}$$

Then the partial product recurrence is

$$\begin{aligned} P_j &= X_{j-1}A_{j-1}2^j + (X_j a_j + A_{j-1}x_j)2^{-j}2^j + B \\ &= 2P_{j-1} + (X_j a_j + A_{j-1}x_j) \end{aligned} \tag{2.4}$$

where $P_0 = B$.

Let $d_i \in \{-1,0,1\}$ be the ith computed product digit. Then

$$w_j = P_j - 2^j D_{j-1}, \quad D_{j-1} = \sum_{i=0}^{j-1} d_i 2^{-i} \tag{2.5}$$

is the jth residual, i.e., the difference between the true and computed partial product. The residual recurrence is:

$$w_j = 2(w_{j-1} - d_{j-1}) + X_j a_j + A_{j-1}x_j \tag{2.6}$$

or, defining the diminished residual $z_j = w_j - d_j$:

$$w_j = 2(z_{j-1}) + X_j a_j + A_{j-1}x_j$$

At step m

$$A_m X_m + B = \sum_{i=0}^{m} d_i 2^{-i} + (z_m)2^{-m} \tag{2.7}$$

The result digit d_j is selected according to the rule [ERCE75]:

$$d_j = sign(w_j) * \left\lfloor |w_j| + \frac{1}{2} \right\rfloor, \qquad |w_j| < \frac{3}{2} \tag{2.8}$$

Consequently, $|w_j - d_j| \le \frac{1}{2}$, and $\sum_{i=0}^{n} d_i 2^{-i}$ represents the most significant half of the product $Y = A_m X_m + B$.

To use carry-save addition we compute $\hat{w}_j$, an approximation to w_j, and choose d_j so that $|\hat{w}_j - d_j|$ is less than $\frac{1}{2}$, where $\hat{w}$ is accurate enough to insure that the actual $|w_j - d_j|$ is less than $\frac{1}{2} + \varepsilon$. This is satisfied if

$$|A|, |X| < \frac{1}{8} \qquad \varepsilon = \frac{1}{8} \tag{2.9}$$

The value of ε means that the fraction part of $\hat{w}$ must be computed to three binary places for selection of d. That is, $|w - \hat{w}| < \frac{1}{8}$. The operands A and X must also be scaled to satisfy the condition (2.9). Since $P_0 = B$, $B < 1/2$.

Figure 1 shows a block diagram of the arithmetic unit corresponding to the recurrence (2.6).

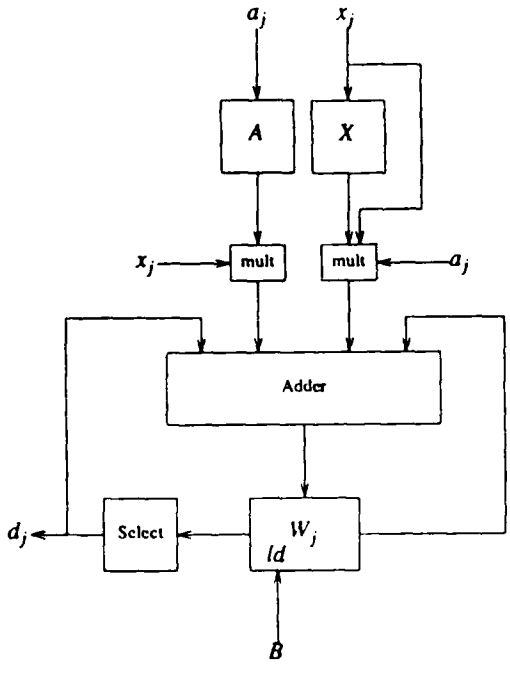

Figure 1. Block Diagram of the On-line Unit

3. The Design

In each cycle, the equation (2.6) is evaluated as follows. The input digits a_j and x_j are appended to the current A and X, followed by the multiplication to produce $X_j a_j$ and $A_{j-1} x_j$. Upon adding these to z_{j-1} to determine w_j, the proper d_j is selected according to formula (2.8). Finally, the result digit is sent out, and $w_j - d_j$ is formed.

The output must be in the signed-digit form, and, consequently, the input format must subsume the signed-digit form. However, a simpler design is obtained by using two's complement arithmetic for internal operations. For this reason, the input operands are converted on-the-fly [ERCE85] from the signed-digit into two's complement form. The speed of the implementation is limited primarily by the time to evaluate the equation (2.6), i.e., the cycle (step) time. Next we discuss our approach in reducing the cycle time by decoupling the digit selection from the residual computation and by introducing pipelining in the step computation. Later in this section we discuss the bit-slice organization of the module. The organization of multi-module units is discussed in [TULL86a].

Pipelining of Addition and Selection

To use carry-free addition in the recurrence equation (2.6), w is represented as the sum of the carry part (C) and the sum part (S). Since the absolute value of w is guaranteed to be less than 3/2 (eq. 2.8), it is sufficient to use two bits to the left of the radix point. We only need add the first 3 bits (positions i = -1, 0, 1) of C and S plus a carry bit from the positions i = 2,3 as shown in Figure 2.

$$\begin{array}{r} C_{-1} C_0 . C_1 C_2 C_3 \\ + \; S_{-1} S_0 . S_1 S_2 S_3 \\ \hline W_{-1} W_0 . W_1 \end{array} \qquad c_{in}$$

Figure 2. $\hat{w}$ Computation

In Figure 2, $c_{in} = C_2 S_2 + (C_2 + S_2) C_3 S_3$ and $\hat{w}$, the truncated value of w ($W_{-1} W_0 W_1$), is simply the sum of the three most significant bits of C and S.

The recurrence equation (2.6) requires that the selected value of d in each cycle is subtracted from w. Since the highest three bits of w must be calculated for selection of d, it is not necessary to subtract d from the carry-sum representation of w. Instead, we calculate $\hat{w} - d$ (since d is 0, 1, or -1 subtracting it from the most significant portion of w will not affect other bits), keep track of that value separate from C and S and set the most significant three bits of C ($C_{-1,0,1}$) and S to 0. The result is a different representation of w that is equal to $z = w - d$.

The cycle time of operation, which limits the maximum speed of the chip, is determined by the recurrence formula and involves two steps of carry-save adders, the computation of $W_{-1,0,1}$ and the c_{in} bit, the selection of d, and, finally, the computation of $\hat{w} - d$. However, note that (i) it is possible to calculate $\hat{w} - d$ before knowing d, and (ii) surprisingly enough, the information needed to compute $\hat{w} - d$ at step i does not depend on the result of $\hat{w} - d$ from step $i-1$.

From the possible ranges of w and the corresponding values of d

$$d = \begin{cases} 1 & \text{if } \frac{1}{2} \le w < 1\frac{1}{2} \\ 0 & \text{if } -\frac{1}{2} \le w < \frac{1}{2} \\ -1 & \text{if } -1\frac{1}{2} < w < -\frac{1}{2} \end{cases} \quad (3.1)$$

we obtain the following table for all possible values of $\hat{w}$ and c_{in} and the resulting values of d and $\hat{w}-d$ (which is t).

Table 1. Digit Selection

$\hat{w}$	c_{in}	d	$\hat{w}-d$
00.0	0	0	00.0
	1	1	11.0
00.1	0	1	11.1
	1	1	11.1
01.0	0	1	00.0
	1	-	----
01.1	0	-	----
	1	-	----
10.0	0	-	----
	1	-1	11.0
10.1	0	-1	11.1
	1	-1	11.1
11.0	0	-1	00.0
	1	0	11.0
11.1	0	0	11.1
	0	0	11.1

The resulting expressions for the estimate of the diminished residual are:

$$t_1 = W_1$$

$$t_0 = t_{-1} = W_1 + c_{in} \quad (3.2)$$

This shows that $t=\hat{w}-d$ is simpler to compute than d, and since only t is critical to the recurrence formula (and thus the principal cycle), we can delay the selection of d until later without affecting the speed of the main cycle. Note that t can be computed without knowing t from the previous cycle. To see this, we must look at the actual computation of $\hat{w}$:

$$\begin{array}{r} z_{-1}\ z_0 \\ C_{-1}C_0.C_1C_2C_3 \\ +\ S_{-1}S_0.S_1S_2S_3 \\ \hline W_{-1}W_0.W_1 \\ c_{in} \end{array}$$

In order for this to accurately represent w, the most significant two bits of C and S must not duplicate the information in t. For that reason, in those bits, the previous C and S are not added in because they are included in t. For example, $2C_0 + S_0 = a_0x_i + x_0a_i + c_{int(1)}$, where $c_{int(1)}$ is just the intermediate carry from bit position 1. Now t, as seen in equation (3.2), only depends on W_1 and c_{in}, which only depend on the previous C_1 and S_1, C_2 and S_2, C_3 and S_3. This implies, then, that once we have produced C and S from the carry-save adders at step i, we have all the information needed to begin computing t in step $i+1$. Theoretically, it could be done without ever computing t at step i.

This allows us to also take the computation of t completely out of the step cycle (as well as the computation of $\hat{w}$ and c_{in} along the way). Therefore, the operation that limits the speed of the step cycle is just two parallel carry-save adder steps (to reduce four summands, $A_{j-1}x_j$, X_ja_j, C, and S to two, C and S). The most significant bits of w are important, however, for the selection of d at each step. The formulas for computing d, based on $\hat{w}$ and c_{in} derived from Table 1 are given below. Since d is in signed-digit format, there are two components of d, a sign component, d_s, and a magnitude component, d_d.

$$d_d = \bar{w}_0\bar{w}_1c_{in} + \bar{w}_0w_1 + \bar{w}_{-1}w_0 + w_0\bar{w}_1\bar{c}_{in} + w_1\bar{w}_0$$

$$d_s = w_{-1}\bar{w}_1\bar{c}_{in} + w_{-1}\bar{w}_0 \quad (3.3)$$

At each step we produce the least significant bits (all those to the right of the radix point) of the results C and S by carry-save addition, and then send those to the next step to be added again to Ax and Xa. At the same time produce incomplete values of the most significant bits of C and S (by assuming that the most significant bits of the last w (C and S) were zero, since they will be added again later). All this information is passed on to the next stage of the pipeline which computes t and d from the information available (that information being the incomplete C and S, the previous $t = z_{-1}$ and z_0), while the previous pipeline stage produces another C and S.

Figure 3 illustrates the pipelined operation of the unit, showing input of the operands, conversion to two's complement representation, multiplication of the vectors by the current digit, the two adders, and finally the computation of z and d, the next output. New digits are input every cycle and similarly a result digit is output every cycle. The first adder step spans two stages. This is necessary because in order to complete the addition it needs C produced by the second adder from the previous trip

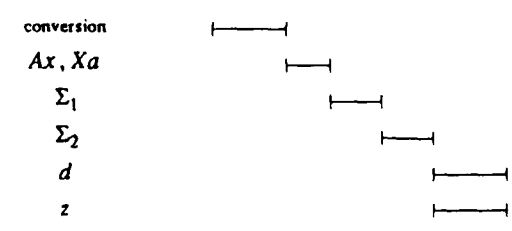

Figure 3. Pipelined Operation of the Unit

through the pipeline. This is not available until the beginning of cycle four; therefore, as much of the addition as possible is done in the previous cycle (generating all possible results from the other two operands so that when C is available it need only choose between them).

Bit-Slice Organization

The module is implemented as a submodule of 8 bit-

slices, selection submodule, and control submodule. The design can be easily expanded to implement wider modules. Each bit-slice consists of five principal sections as shown in Figure 4.

A-X register section holds the values of A and X and also contains the signed-digit-to-two's-complement conversion logic. There is also a traveling load signal across the slices to control the loading of the current a_j and x_j at the correct slice. Because of the nature of the recurrence relation, it is important that a_j be loaded one cycle after x_j so that A is actually A_{j-1} as in the recurrence equation.

Multiplier section implements the multiplications $X_j a_j$ and $A_{j-1} x_j$. This is a multiplexer which selects the proper multiple of each bit (e.g., it selects x_i, 0, or the complement of x_i depending on whether a_j is 1, 0, or -1). There is also some logic in the low order module to compensate for the complementation in two's-complement arithmetic.

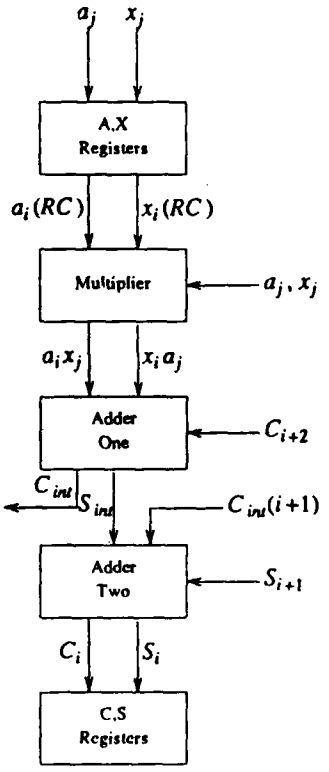

Figure 4. Organization of a Bit Slice

Carry-save adders section consists of two carry-save adders. The first one adds the two outputs of the multiplier (Ax and Xa for short) with C, the carry part of the output of the final carry-save adder from the previous cycle. These produce two intermediate outputs, S_{int} and C_{int}. These two are added (with C_{int} shifted) in the second carry-save adder to S and produce the two components of w, C and S.

The last section of the bit slice contains the C and S registers. The outputs of the adders are shifted to the correct slice (at which they will be used in the next cycle) and stored in registers. Initially the S registers are set to 0 and the C registers are set to the value of B (B is available off-line and in two's complement form).

A module contains 8 bit-slices operating in parallel. However, each also contains two other bit-slices that are only used if the module is in the most significant position. These are the two bits to the left of the radix point. These two bit positions are necessary to accommodate the maximum possible absolute value of w, 3/2, which can be represented in two's complement with two bits to the left of the radix point. They are similar to the other bit-slices, except for a few minor changes.

The *selection submodule* chooses d and subtracts it from the most significant bits of w. It takes the inputs C_{-1} to C_3, S_{-1} to S_3 (10 inputs) and z_{-1}, z_0 from the previous cycle and then produces d, the current output, and z to be used in the next cycle. Then, with those results, d and z can be determined according to equations (2.6) and (3.1). The organization of the selection section is shown in Figure 5.

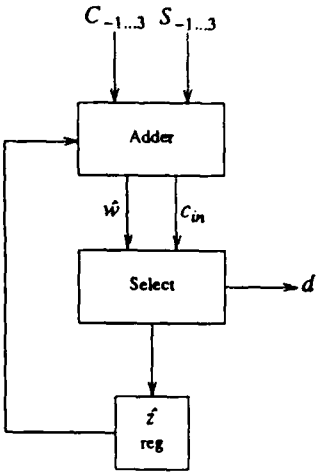

Figure 5. Selection Section Organization

Although results of the bit slice operation are needed for selection, the bit-slice operation is not dependent upon any results of the selection process as illustrated in Figure 5. The adder in this figure could more accurately be considered two adders in parallel, one for $\hat{w}$ and one for c_{in}.

4. The Chip Characteristics

The 8-bit chip, implemented in 4μ nMOS (the minimum feature size is 4 μ), measures 1,866 μ by 1,838 μ without pads. It contains 1,957 transistors. With the input and output pads the area is increased to 2,646 μ by 2,568 μ (shown in Figures 7 and 8).

The corresponding measurements for the 16-bit module chip are 3,002 μ by 1,838 μ without pads, 3,694 μ by 2,568 μ with pads, and a total of 3,165 transistors.

For any size chip there is a fixed area of 730 by 1,838 μ

plus 142 by 1,838 μ for each bit-slice. In other words, the size of a b-bit module would be $730+b*142$ μ by 1,838 μ. With pads, it is less exact, but it indicates an overhead of about 1,598 μ by 2,568 μ plus about 132 μ by 2,568 for each bit-slice. At the transistor level, there is a fixed section of 749 transistors (mostly the selection logic) plus an additional 151 transistors per bit-slice. The layout of a bit-slice is shown in Figure 6.

Each bit-slice (151 transistors) is composed as follows: the largest portion, 60 transistors, implements the signed-digit to range complement conversion and A and X registers. The multiplier uses 16 transistors, all enhancement pass transistors. The two carry save adders combined represent 55 transistors. Finally, the C and S registers are formed from 20 transistors. Each bit-slice is comprised of 27 logic "gates," determined by counting the number of depletion mode transistors. The rest of the transistors are enhancement mode transistors that are either part of a logic gate or are pass transistors. Two PLA's that implement the selection logic combine for 168 transistors, or more than 20% of the 749 transistors in the fixed section. The two extra bit-slices to the left of the radix point account for another 212 transistors, or another 30% of the fixed section. The rest of the transistors are used for input and output buffering, and control signal generating and routing.

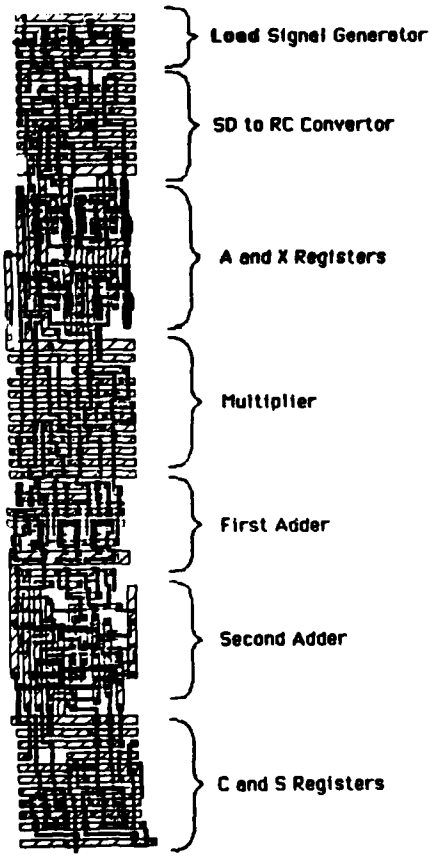

Figure 6 -- Layout of a Bit Slice

Of the area actually used (i.e., ignoring empty spaces), approximately 43% of it was used by the 8 bit-slices, 28% was used for the various control signals and on- and off-chip communication, 19% for the digit selection logic (this is rather high because it was done with PLA's), and 9% was used by the two bits of sign extension. Of course, that is for the 8-bit module. For the 16-bit modules, the 16 bit-slices represent 60% of the useful area.

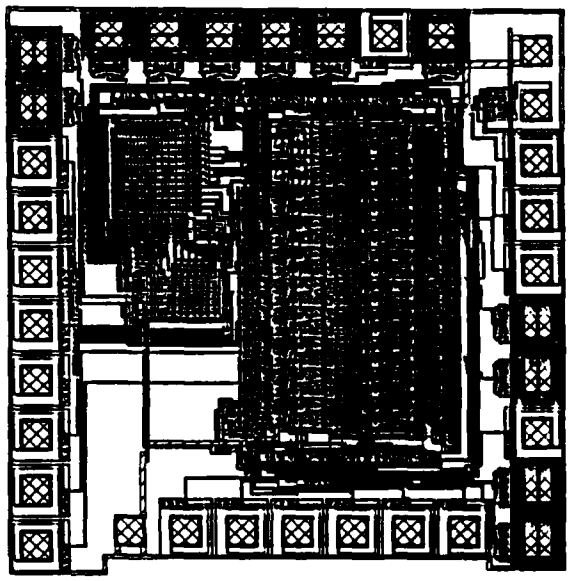

Figure 7 -- Layout of the Chip

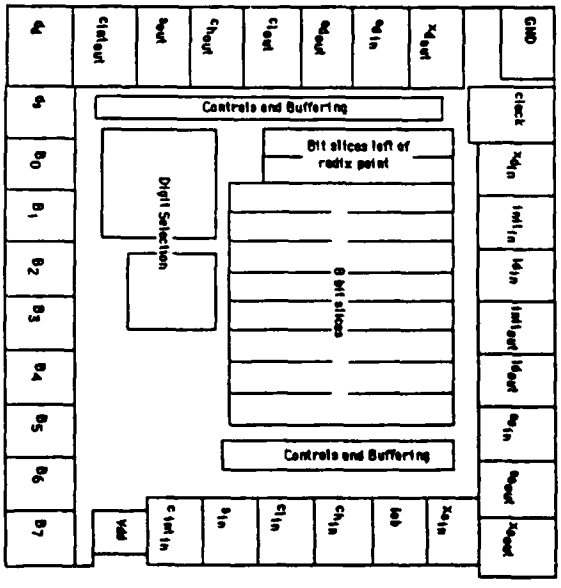

Figure 8 -- Floorplan of the Chip

Timing Analysis

The simulations indicate that the chip will operate in a pipelined manner with a maximum clock cycle of about 110 nanoseconds in the 4-micron nMOS implementation. For a less technology-dependent comparison, that figure corresponds to three gate delays per clock cycle, plus a few pass transistors that contribute a small percentage of that delay. These results were obtained by using Mextra [MAYO83], a circuit extractor, and Crystal [OUST83], a timing analysis program which finds the worst case path in a circuit. The longest pipeline stage, which identifies the maximum speed of the circuit is found by identifying the longest path between two successive ϕ_1 or two successive ϕ_2 signals, which may be the same signal in circular circuitry. In this case the worst case path travels through the C register (loaded at ϕ_2), enabling S_{inf}, which is input to the adder gate that produces S to be loaded in the S register at ϕ_2. The signal path travels through three gates, two for the register and one for the adder, plus a few pass transistors. Crystal was used to find slowest paths and SPICE was used to verify maximum cycle times. The chip design was also simulated in 1-micron technology. Although no accurate SPICE parameters were known, the worst case delays were found through the circuit extractor and Crystal. In this case it was found that the chip would be able to operate at a cycle time of less than 10 nanoseconds.

Functional Design Checking

In order to check that the chip operates correctly functionally and logically, a couple of tools were used. These were 1) a program in C to simulate the desired operation of the chip, implementing the bit-level algorithm of the on-line module and 2) the switch-level logic simulation program Esim [MAYO83] which takes as input the results of the circuit extractor Mextra. In other words, Esim simulates the actual circuit as defined by the VLSI artwork.

The first step was to test the C program to ensure that it produced the desired correct results in all cases. After this was done, this program produced results that could be checked against other simulations for both debugging purposes and verification. The next step, then, involved comparing Esim results with those predicted by the C program for particular inputs. This was done quite carefully, checking not just outputs, but many intermediate results within the circuit. After this, the circuit was again simulated by Esim, but now paying attention primarily to the inputs and outputs and checking them for accuracy. This was done for a large variety of inputs.

All simulation up until this point had been assuming that the unit consisted of only one module, in other words that the operands were only eight bits wide and therefore there was no concern about intermodule communication. The next step was to modify the C program to operate with the same algorithm, but now 16 bits wide. Then the circuit was simulated, first acting as the low order byte and keeping track of all the outputs to the higher order byte (also again checking intermediate values as well against the correct bits in the C simulation). Next the high byte was simulated with the outputs of the low byte as inputs. The outputs of this module were then checked to be accurate. In this way the correctness of the module was verified in all special cases and positions that it could occupy within a unit.

Several of these results are included in [TULL86b]: the C simulation program, results of this simulation for three different inputs, and Esim outputs on the identical inputs. The results of the two simulations can therefore be cross-referenced. In addition, Crystal and SPICE outputs are included to verify timing results.

5. An Example

In order to illustrate the use and some of advantages of the on-line unit, we give an example. Consider finding a root of a polynomial equation by an iterative method. For instance, the root of a fourth degree polynomial (equation 5.1) could be found by solving for x iteratively.

$$x = p_4x^4 + p_3x^3 + p_2x^2 + p_1x + p_0 \qquad (5.1)$$

The parameters of this equation and the initial value of x are assumed to satisfy required convergence conditions. The polynomial equation is evaluated using the method given in [ERCE75, ERCE77]. The configuration in Figure 9 utilizing buffers for x would solve equation 5.1 iteratively and continuously, once every n cycles, where n is the length of operands (or desired output). In Figure 9, n is assumed to be 32, again using 16-bit modules, resulting in a six cycle latency between one unit and the next.

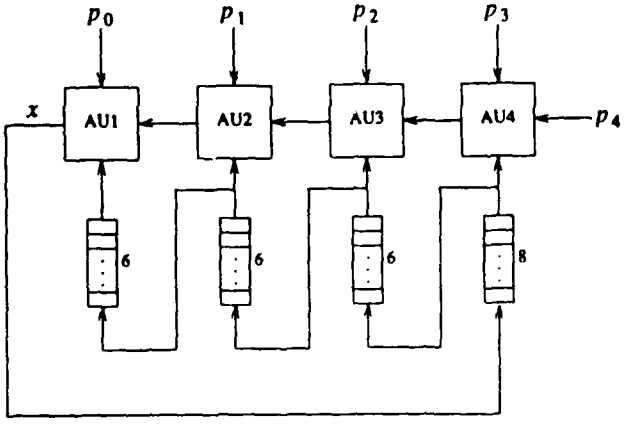

Figure 9. Solving a Polynomial Equation

In this configuration, the new x would be produced beginning 24 cycles after the old is input to module AU4. After eight more cycles AU4 would be ready to begin to input the new x, which it could do without influencing the outputs at AU1 before it finished producing all 32 digits of the current x. The timing of the four units in terms of their inputs is shown in Figure 10.

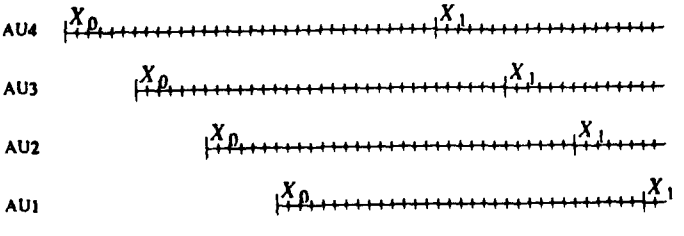

Figure 10. Timing of Root Finder

This would be possible because when a new *init* signal arrives at a 2-module unit, the next five outputs will remain unaffected and the sixth will be the first digit of the next output. This means that even if x_{32} and $p_{4_{32}}$ are followed immediately by *init* and x_1 and p_{4_1} (which are always input simultaneously), y_{32} of AU4 will be produced five cycles later unaffected, followed by the next y_1 on the sixth. The same is true for each module so that AU1 will produce the new x_1 immediately following the last x_{32}. In this manner, this configuration after an initial setup time of 23 cycles (four times the latency minus 1), will then complete an iteration of the fourth degree polynomial every 32 cycles. In that case, for instance, 10 iterations would require 343 clock cycles. In comparing this scheme with one using conventional arithmetic chips several differences are seen. One is that although four different units are operating in a pipelined manner on the output of the pipeline, they are all operating 100% of the time. With conventional methods, none of the units could operate until the previous one completed, and the whole process could not be restarted until AU1 completed. Therefore, each of the modules would be operating only 25% of the time.

Another difference is the small number of connections between units (and modules within each unit). There are only two lines between each unit in this configuration, and there is never any single line having to broadcast signals. With conventional chips, there would have to be 32 lines between each two chips for the pipeline to run as efficiently as possibly.

Each of the on-line clock cycles corresponds to approximately one carry-save adder step and a register. A 32-bit module to compute $AX + B$ in the conventional manner would require 32 similar clock cycles, or if some simple radix-4 recoding was used, it would require 16 add-store steps. If computed with a similar configuration, these modules would then require 64 clock cycles, because no overlap would be possible. To minimize delay, a tree-like structure could be used to evaluate the polynomial in three steps rather than four as shown in Figure 11 (Estrin's method).

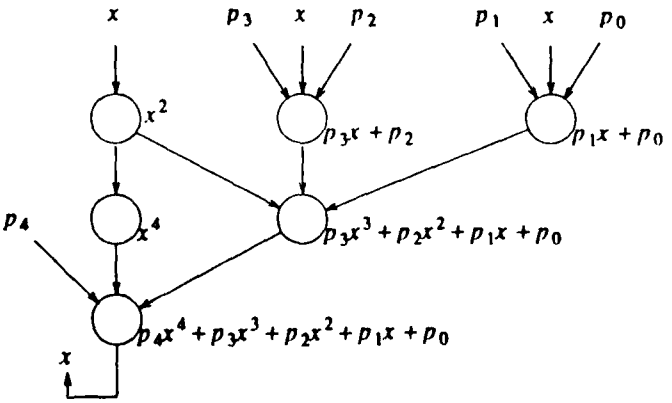

Figure 11. Conventional Root-Finding Solution of $P_4(x)$

In this case, the time to complete one iteration would be 48 steps. Since the next iteration could not be started until the previous one completed, the time for 10 iterations would be 480 clock cycles.

In addition, the performance of the on-line method can be improved by adding more hardware. If we duplicated the hardware of Figure 9, the output of the first polynomial evaluator could be sent as input to the second, to avoid the 8-step delay as x waits in the buffer for the pipeline to be ready for new inputs. In this way, the second could begin immediately after 24 cycles accepting the new x, and 24 cycles later, it would produce the first bit of x_2 to be input to the free first evaluator. Thus, with multiple on-line evaluators, an iteration could be begun every Δ steps, where Δ is the total latency of one evaluator. Therefore, with this scheme, 10 iterations could be completed in 271 cycles. This improvement is even greater when n, the width of the operands, is much larger than Δ, as is the case when $n = 64$, resulting in $\Delta = 32$. With 64-bit operands, it could still be done at maximum speed with two evaluators (because Δ is exactly half of n), but with any larger operands, more than two evaluators would be necessary. The total delays for N iterations of operands M bits wide for the conventional (conv), on-line (ol), and on-line with multiple evaluators (olm) schemes are:

$$D_{conv} = \frac{3NM}{2}, \qquad D_{ol} = NM + \Delta - 1, \qquad D_{olm} = N\Delta + M - 1$$

Table 2 summarizes the time to complete the evaluation for the three different schemes varying the number of iterations and the size of the operands. From this it can be seen that with multiple on-line configurations, the speedup approaches a maximum of 3 in the 64-bit case.

Table 2. Comparison of Root Solving Implementations

Number of Iterations	32-bit Operands			64-bit Operands		
	Conv	On-line	Multiple On-line	Conv	On-line	Multiple On-line
1	48	55	55	96	91	91
10	480	343	271	960	667	383
100	4800	3223	2431	9600	6427	3263

Summary

A design and VLSI NMOS implementation of a radix-2 on-line algorithm for computing $AX+B$ are described. The design is characterized by a very short cycle time of about three gate delays, achieved through internal pipelining. The chip consists of a group of bit-slices (8+2), digit selection logic, and control. In the 4μ technology the design occupies an area of 2,646×2,568 μ^2 (including I/O pads) and is expected to have a 110 ns cycle. In the 1 μ NMOS technology, the cycle time is estimated to be less than 10 ns.

Ackowledgements This research has been supported in part by the Office of Naval Research Contract No. N00014-85-K- 0159 "On-Line Arithmetic Algorithms and Structures for VLSI". We are grateful to Prof. Tomas Lang of UCLA for helpful suggestions and interest.

References

[AVIZ61] A. Avizienis, *Signed Digit Number Representations for Fast Parallel Arithmetic*, IRE Transactions on Electronic Computers, 1961, pp. 389-400.

[ERCE75] M. D. Ercegovac, *A General Method for Evaluation of Functions and Computations in a Digital Computer*, Ph.D. Thesis, Report No. 750, Department of Computer Science, University of Illinois, Urbana, August 1975.

[ERCE77] M.D. Ercegovac, *A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer*, IEEE Transactions on Computers, Vol. C-26(7), 1977, pp. 667-680.

[ERCE84] M.D. Ercegovac, *On-Line Arithmetic: An Overview*, Proceedings SPIE Conference on Real-Time Signal Processing, San Diego, 1984, pp.86-92.

[ERCE85] M.D. Ercegovac and T. Lang, *On-the-Fly Conversion of Redundant into Conventional Representations*, Report No. CSD-850026, UCLA Computer Science Department, August 1985.

[MAYO83] R. N. Mayo, J. K. Ousterhout, W. S. Scott, editors, 1983 VLSI Tools Report No. UCB/CSD 83/115, Computer Science Department, University of California, Berkeley, March, 1983.

[MEAD80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[NAGE73] L. Nagel, D. Pederson, *Simulation Program with Integrated Ciruit Emphasis (SPICE)*, 16th Midwest Symposium on Circuit Theory, Waterloo, Ontario, April 12, 1973.

[OUST81] J. K. Ousterhout, *Caesar: An Interactive Editor for VLSI*, VLSI Design, Vol II, No. 4, Fourth Quarter, 1981, pp. 34-38.

[OUST83] J. K. Ousterhout, *Crystal: A Timing Analyzer for nMOS VLSI Circuits*, Third Cal Tech Conference on Very Large Scale Integration, 1983, pp. 57-69.

[TULL86a] D.M. Tullsen, *A Very Large Scale Integration Implementation of an On-line Arithmetic Unit*, MS Thesis, UCLA Computer Science Department, June 1986.

[TULL86b] D.M. Tullsen, *Simulations of an On-Line Arithmetic Unit Design*, Internal Memorandum, UCLA Computer Science Department, May 1986.

[TRIV77] K.S. Trivedi and M.D. Ercegovac, *On-Line Algorithms for Division and Multiplication* IEEE Transactions on Computers, Vol. C-26, No. 7, July 1977.

1988 SPIE CONFERENCE
SAN DIEGO, CA Aug.88

Implementation of an SVD Processor Using Redundant CORDIC

Miloš D. Ercegovac and Tomas Lang
UCLA Computer Science Department, University of California, Los Angeles

Abstract. An implementation of the diagonal and off-diagonal processors for an array performing the singular value decomposition (SVD) is presented. The implementation uses a modification of the CORDIC module that utilizes carry-save addition instead of carry-propagate addition, resulting in a significant improvement in speed. Moreover, the calculation of the angles and of the two-sided rotation are overlapped. To achieve this overlapping, the calculation of the rotation angles includes an on-line module. Finally, the carry-save calculation and the overlapping result in a variable CORDIC scaling factor. This factor is computed and the correction performed by on-line division. Pipelining and rotation interleaving are used to reduce the implementation complexity. The speed is evaluated and compared with that obtained when conventional CORDIC modules are used.

1. Introduction

Many compute-intensive applications include matrix computations that involve the calculation of angles and their use in rotations. Examples are matrix triangularization and singular value decomposition (SVD) [GOLU83]. To achieve adequate throughput, parallel structures have been proposed which are typically organized in linear, triangular, or square arrays [GENT81, CIMI81, AHME82, LUK86, CAVA87]. The angle(s) are computed in boundary (diagonal) processors and broadcast to other processors for rotation.

A possible implementation of the angle calculations and of the rotations is to use a standard arithmetic processor that performs the basic operations of addition, multiplication, and, perhaps, division. In such a case, the required calculation are implemented as sequences of the basic operations. However, because of the lengthy sequences required, the resulting implementation is slow. Consequently, it it of interest to develop special-purpose processors for these applications, especially because the design of these application-specific chips is becoming cost effective.

In this paper we concentrate on the implementation of the SVD because of its interest and because of its challenging complexity, which is attractive to illustrate the advantages of the application-specific approach. For the definition of the SVD, the basic algorithms for its computation, and its applications, the reader is directed to [GOLU83] and [LUK86].

Several alternative implementations are possible for the calculation of the required angles and the execution of the rotations. Of particular interest are the following two:

a) The *sine* and *cosine* of the angles are computed by means of a sequence of operations involving squaring, addition, multiplication, square root, and division. The rotation is then done by several multiplications and additions. The main advantages of this approach are that efficient implementations for the primitive operations are known and that redundancy can be used to improve the speed [ROB58, AVI61, ATKI75]. However, it requires various different modules and consists of several dependent computations. To reduce the delay introduced by this, it is possible to use the on-line approach, which allows the overlapping of these dependent operations; examples of this have been presented in [ERCE87a] and [ERCE87b].

b) Directly calculating the angles using CORDIC operations [VOLD59, WALT71] and using the same approach for the rotation. This method has been first proposed for matrix triangularization in [AHME82] and for the singular value decomposition in [CAVA87]. It has as advantage that a small number of operations is required and that the same module can be used for both the angle calculation and the rotation. However, the conventional implementation of the CORDIC module has two disadvantages: it is slow, because it involves recurrences including carry-propagate addition and variable shifting, and area-consuming because of the need for variable shifters and ROMs to store angle constants.

We present here an implementation for SVD that uses a combination of redundant CORDIC modules together with other on-line modules. The resulting implementation is significantly faster than the one using conventional CORDIC modules, because of the elimination of carry-propagate adders, and the overlapping of operations made possible by the on-line approach.

To make the speed comparisons meaningful a suitable measure has to be used. In some studies the comparisons are done in terms of the number of addition-like steps, which appear as basic components in the iterations for operations such as multiplication, division, and CORDIC. However, this is not an adequate measure since the time for addition depends on the type of addition performed. More specifically, the time of carry-propagate addition is several times larger than that of redundant (carry-save or signed-digit) addition. It might be claimed that this does not change the validity of measuring in terms of additions, since for a particular implementation the corresponding type of addition would be used. However, this is not correct since not all algorithms can be directly transformed from one using carry-propagate addition into one using redundant additions. Furthermore, when fast redundant additions are used, other terms which were neglected when carry-propagate additions are considered become important. As a consequence, to make more meaningful comparisons, we define a basic clock cycle and estimate the time of the various operations in terms of this clock cycle.

The implementations we describe are for floating-point representations since this format provides better numerical characteristics and results in a system which is easier to use in a variety of environments than the fixed-point alternative. We use the characteristics of the algorithm to reduce the additional overhead introduced by the floating-point representation.

In this paper we concentrate on the implementation issues. The more theoretical aspects of the development of the algorithms are presented in [ERCE87d].

2. Parallel Implementations of Singular Value Decomposition (SVD)

Because of the computation-intensive nature of the algorithms for SVD, great interest has appeared on parallel arrays, as discussed in [BRENT85a], [BRENT85b], [LUK86], and [CAVA87]. The primitive operation in these cases is the diagonalization of a 2x2 matrix by the rotations $R(\theta_l)$ and $R(\theta_r)$ (using the notation in [CAVA87]), such that

$$R(\theta_l)^T \begin{bmatrix} a & b \\ c & d \end{bmatrix} R(\theta_r) = \begin{bmatrix} e & 0 \\ 0 & f \end{bmatrix} \tag{1}$$

where θ_l and θ_r are the left and right rotation angles, respectively. The corresponding rotation matrix is

$$R(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{2}$$

Several methods can be used to perform these rotations, in particular the two-step method and the direct two-angle method [BREN85]. Because of the use of CORDIC, we consider here the latter method.

With respect to the implementation, two approaches are possible: a) the computation of $\cos\theta$ and $\sin\theta$ by a sequence of primitive operations, such as squaring, division, and square root, or b) the use of the CORDIC procedure for direct computation of the angles and of the rotations. An implementation using the first approach using the two-step method is reported in [BREN85] and of the second approach using the two-angle method in [CAVA87]. In [CAVA87] a comparison of these two implementations is made, in terms of area and time complexity.

In this paper, we use redundant CORDIC and on-line modules to improve the speed of execution of the two-angle algorithm.

Direct two-angle algorithm for SVD using CORDIC

Figure 1 shows the dependencies of the algorithm. First, two concurrent CORDIC circular operations compute the angles

$$\theta_s = \tan^{-1}\left(\frac{c+b}{d-a}\right) \quad \theta_d = \tan^{-1}\left(\frac{c-b}{d+a}\right) \tag{3}$$

STEP 1 compute $p = c+b$; $q = c-b$; $s = d+a$; $t = d-a$

STEP 2 CORDIC 1: (p,t) produce Θ_{sum} ; CORDIC 2: (q,s) produce Θ_{dif}

STEP 3 compute $\Theta_l = (\Theta_{sum} - \Theta_{dif})/2$; $\Theta_r = (\Theta_{sum} + \Theta_{dif})/2$

STEP 4 CORDIC 3: rotation Θ_l

STEP 5 CORDIC 4: rotation Θ_r

STEP 6 SCALE FACTOR CORRECTION

Figure 1: CORDIC Scheme for SVD

Then, the two angles θ_l and θ_r are obtained as

$$\theta_l = \frac{(\theta_s - \theta_d)}{2} \quad \theta_r = \frac{(\theta_s + \theta_d)}{2} \tag{4}$$

These angle computations are performed in the angle module (Figure 2). The resulting angles are used for the two-sided rotation that diagonalizes the 2x2 matrix (rotation module). The angles are also sent to the corresponding row and column to produce the two-sided rotation of the rest of the 2x2 matrices.

In [CAVA87] this scheme is implemented quite directly using CORDIC modules (except for the correction factor, which is incorporated in the rotations). This results in a time of $3.25T_c$, each T_c corresponding approximately to n carry-propagate additions, where n is the number of bits of the operands. In [ERCE87b] we proposed modifications to the implementation that improve the speed by a factor of approximately 2.5, mainly because of the overlap between the angle calculation and the rotation and the use of on-line CORDIC for the rotations. However, the speed is still basically dependent on the time to perform a carry-propagate addition of n bits. Here we use the redundant adder version of the CORDIC operation to further improve the speed.

Fast Implementation

The fast implementation of the CORDIC scheme is based on the following two elements:

• The overlap between the angle calculation and the two-sided rotation. This overlap is possible if the angle module produces the angle in a decomposed form suitable to be used directly in the CORDIC steps of the rotations. This is the case, for example, in matrix triangularization and was used in [DEPR84] and [ERCE87d]. In the SVD case, the situation is more complex since the addition and subtraction of expression (4) makes the resulting values not suitable for direct rotation. In particular, if conventional CORDIC modules are used, the values are in the set {-1,0,1} which produces variable scaling factors for the rotations. In [ERCE87b] we proposed a solution to this problem, which consists of calculating the scaling factors and applying the correction by on-line division. Another solution is proposed in [DELO87]; however, this complicates significantly the recurrences and would make them slower, especially when using carry-save adders. In this paper, where we use redundant CORDIC, the values are in the set {-1,-1/2,0,1/2,1}; this requires and additional decomposition, as discussed in Section 4.

• The use of redundant CORDIC in the calculation of angles and rotations. This reduces the CORDIC step time and improves the overall speed.

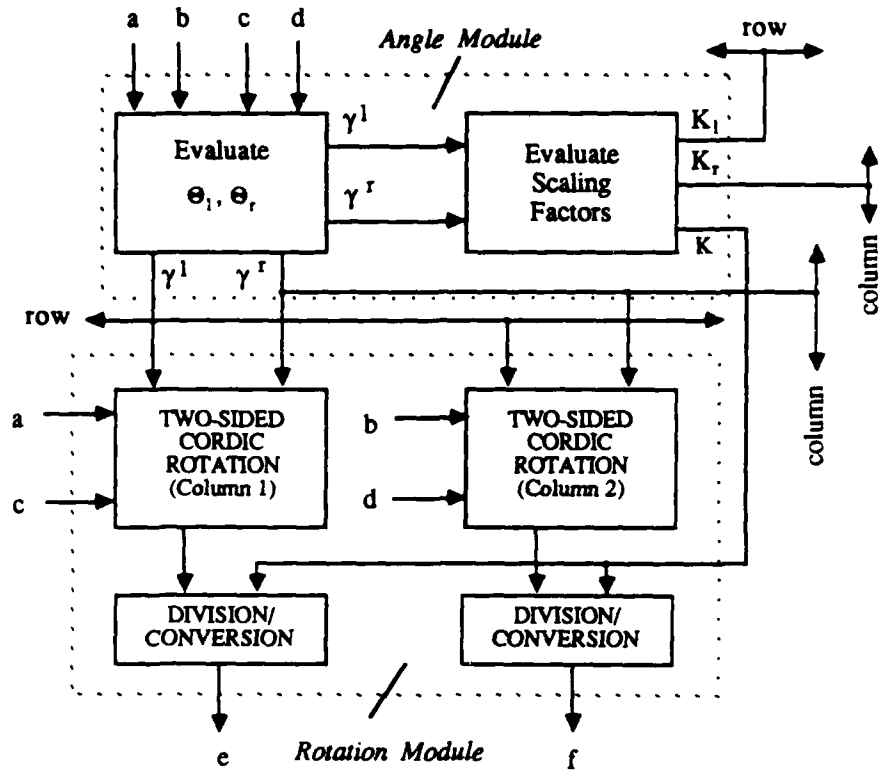

Figure 2a: Diagonal Processor Organization

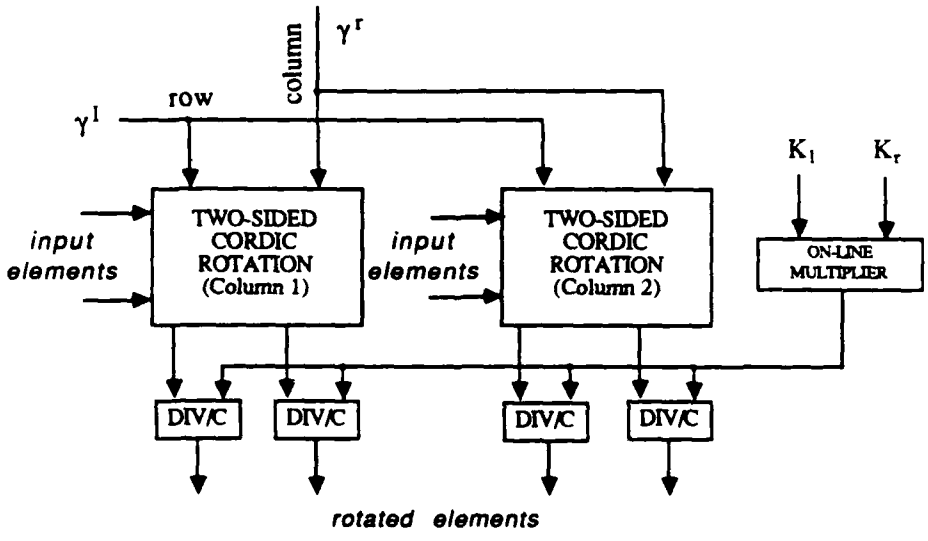

Figure 2b: Off - Diagonal Processor Organization

Critical path and Rotation Interleaving

In addition to these two elements, we look at the critical path of the computation to select implementations for the modules that are balanced to achieve the fastest execution with the lowest complexity. From the timing diagram of Figure 3a, we see that the overlapping between the angles and the two-sided rotation, makes the latter the critical component. This two-sided rotation takes $2n$ CORDIC iterations. On the other hand, the angles can be computed concurrently, so they take just n CORDIC steps. As a consequence, it would be possible to implement the angle module using a CORDIC step twice as slow as the the step for the rotation. However, as indicated in Figure 3a, the calculation of both angles θ_s and θ_d have to be overlapped with the left-angle rotation, since they are both needed to compute θ_l; the angle θ_r, which is also computed from θ_s and θ_d, would be stored until the right-angle rotation. On the other hand, it is possible to balance the computation of the angles and of the rotations if the two rotations are interleaved, performing step j of the left-angle rotation followed by the corresponding step of the right-angle rotation. This is possible since the CORDIC steps are just primitive rotations that can be performed in any order. The timing diagram of Figure 3b, shows that in this case the angle step can be two times slower than the rotation step. We make use of this in the implementations in the following sections.

Pipelining or Module replication

Since the CORDIC step consists of a variable shift and an addition (carry-save in our case, as we will see in the next section) this step can be pipelined. The pipelining does not increase the overall time, it just reduces the clock period. This pipelining is only useful if independent computations can make use of it. For the computations required for a 2x2 matrix, these independent computations exist: the two angles θ_s and θ_d, for the angle module, and the two-sided rotation of the two columns of the matrix. Consequently, we will describe this pipelined implementation.

In the pipelined implementation the clock period is very small (approximately a 4-2 adder plus a multiplexer plus register loading). This requires a very fast clock. If this clock is not suitable for the particular implementation, it is possible to achieve the same speed with a clock two times slower if a non-pipelined implementation is used and the modules for the angle and for the rotations are replicated. If the clock is still too fast, several steps of the CORDIC operation can be unfolded; this would result in the same overall speed, but with an increase in the amount of hardware.

3. Computation of θ_s and θ_d using Redundant CORDIC

We present an implementation that uses redundant CORDIC operations to calculate the angles θ_s and θ_d, with the resulting improvement in speed because of the smaller addition time.

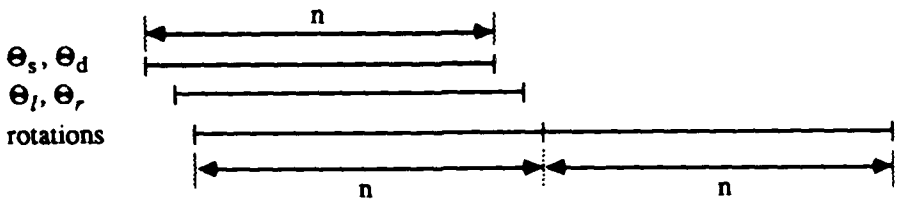

Figure 3a. Critical Path

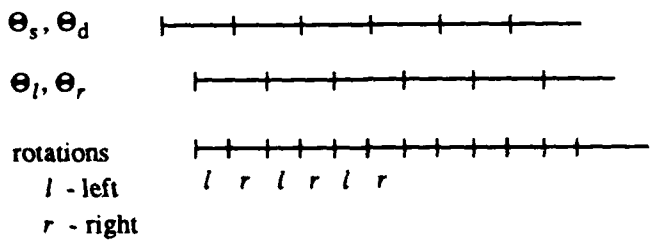

Figure 3b. Interleaving the Rotations

The CORDIC scheme [VOLD59, WALT71] can be used to compute the angle θ, such that $\theta = \tan^{-1}(\frac{A}{B})$. We perform the calculation by a modification of the conventional CORDIC procedure, which requires one shifter instead of two [ERCE87b] and uses a carry-save addition instead of the slower carry-propagate addition. The modified recurrences are

$$x_a[j+1] = x_a[j] + \sigma_j 2^{-2j} w[j] \quad w[j+1] = 2(w[j] - \sigma_j x_a[j]) \quad z_a[j+1] = z_a[j] + \sigma_j \tan^{-1}(2^{-j})$$

The use of carry-save addition instead of carry-propagate addition requires that the determination of σ_j uses an estimate of $w[j]$ instead of its fully assimilated value. To make this possible, it is necessary to produce a redundant representation of θ in terms of the σ_j's. This is achieved by allowing σ_j to take values from the set $\{-1,0,1\}$ instead of from the set $\{-1,1\}$, which is the one used in conventional CORDIC.

The corresponding selection function for σ_j using this redundant set is described in [ERCE87d]. To have the required overlap between the allowed selection intervals it is necessary to normalize $x[j]$. This normalization is obtained directly by the alignment required for floating-point representation, while for fixed-point representation (fractional representation for A and B) the normalization can be performed by scaling both A and B.

The specific selection function for the carry-save form is

$$\sigma_j = \begin{cases} 1 & \text{if } \hat{w}[j] \geq 0 \\ 0 & \text{if } \hat{w}[j] = -1/2 \\ -1 & \text{if } \hat{w}[j] \leq -1 \end{cases}$$

where $\hat{w}[j]$ is an estimate of $w[j]$ with a precision of 1 fractional bit.

Since the angles are used to compute θ_l and θ_r and these angles produce the two-sided rotation, it is not necessary to have the angles in the accumulated form but they can be kept in the decomposed form, that is, they can be represented by the vectors of σ's. This approach has the advantages of eliminating the need for the z recurrence and permits the overlap of the rotations with the angle calculation. This approach was previously used for the simpler application of triangularization [DEPR84] and extended to SVD in [ERCE87b], where the complication of the resulting digit set of $\{-1,0,1\}$ was handled by an on-line computation of the scaling factors. We also use this approach here and discuss the additional problem involved in the next section.

The implementation of the corresponding recurrences using the carry-save approach is shown in Figure 4. To compute the two angles θ_l and θ_r, two modules, operating concurrently, could be used. However, to reduce the hardware needed it is possible to use one pipelined unit, without any speed degradation.

The corresponding unit consists of registers, two 4-2 carry-save adders, a double shifter (for the sum and carry components), and a σ-selection block. The hardware can be further reduced by performing the computation of the angles in the sequential and overlapped manner shown in Figure 5. The corresponding unit contains two 3-2 carry-save adders (instead of 4-2) and one simple shifter (instead of double). The basic clock cycle (or stage delay) corresponds approximately to the maximum of the delay of a 3-2 carry-save adder and of a shifter. As shown in the timing diagram of Figure 5, the consecutive components of each of the angles are produced four clock cycles apart. As mentioned in Section 2, this does not increase the critical path.

Floating-point representation

The implementation can use floating-point representations, as described in [ERCE87d]. The only additional requirement is an initial alignment of $x[1]$ and $w[1]$ so that $x[1]$ is normalized.

4. On-line Computation of θ_l and θ_r

The angles θ_l and θ_r have to be computed by expression (4). The simplest way to do this is to compute σ_j^l and σ_j^r directly from σ_j^s and σ_j^d. The corresponding relations are

$$\sigma_j^l = \frac{\sigma_j^s - \sigma_j^d}{2}, \qquad \sigma_j^r = \frac{\sigma_j^s + \sigma_j^d}{2}$$

resulting in the following table:

σ_j^s	1	1	1	0	0	0	-1	-1	-1
σ_j^d	1	0	-1	1	0	-1	1	0	-1
σ_j^l	0	1/2	1	1/2	0	-1/2	-1	-1/2	0
σ_j^r	1	1/2	0	-1/2	0	1/2	0	-1/2	-1

In contrast with the implementation discussed in [ERCE87b], it is not possible to use directly these values of σ_j^l and σ_j^r for the rotations because of the values ±1/2, which do not lead to a simple rotation step. Because of this, we use these values to compute another decomposition with the digit set {-1,0,1}. That is, we compute the sequences of γ_j^l and γ_j^r such that

$$\theta_l = \sum_{j=0}^{n-1} \sigma_j^l \cdot \tan^{-1}(2^{-j}) = \sum_{j=0}^{n-1} \gamma_j^l \cdot \tan^{-1}(2^{-j}) \quad \sigma_j^l = \{-1,-1/2,0,1/2,1\} \quad \gamma_j^l = \{-1,0,1\}$$

$$\theta_r = \sum_{j=0}^{n-1} \sigma_j^r \cdot \tan^{-1}(2^{-j}) = \sum_{j=0}^{n-1} \gamma_j^r \cdot \tan^{-1}(2^{-j}) \quad \sigma_j^r = \{-1,-1/2,0,1/2,1\} \quad \gamma_j^r = \{-1,0,1\}$$

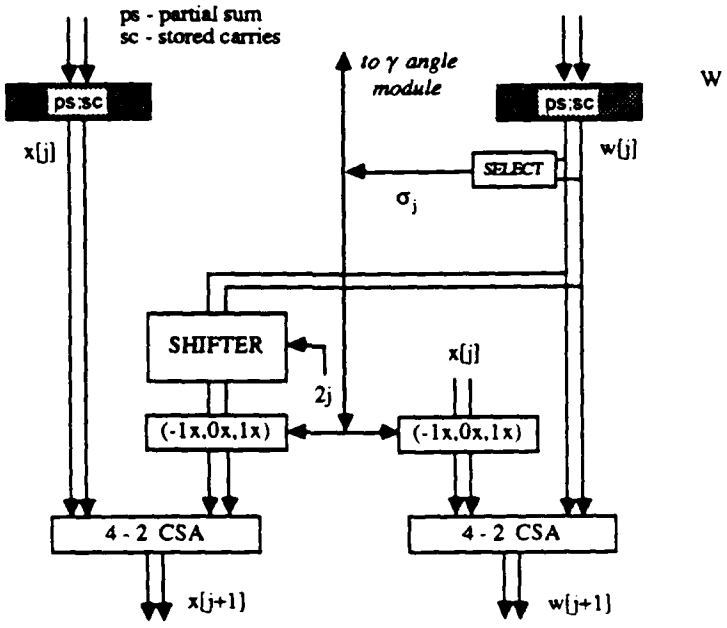

Figure 4. Non-pipelined Implementation for Θ_r Θ_d Calculation

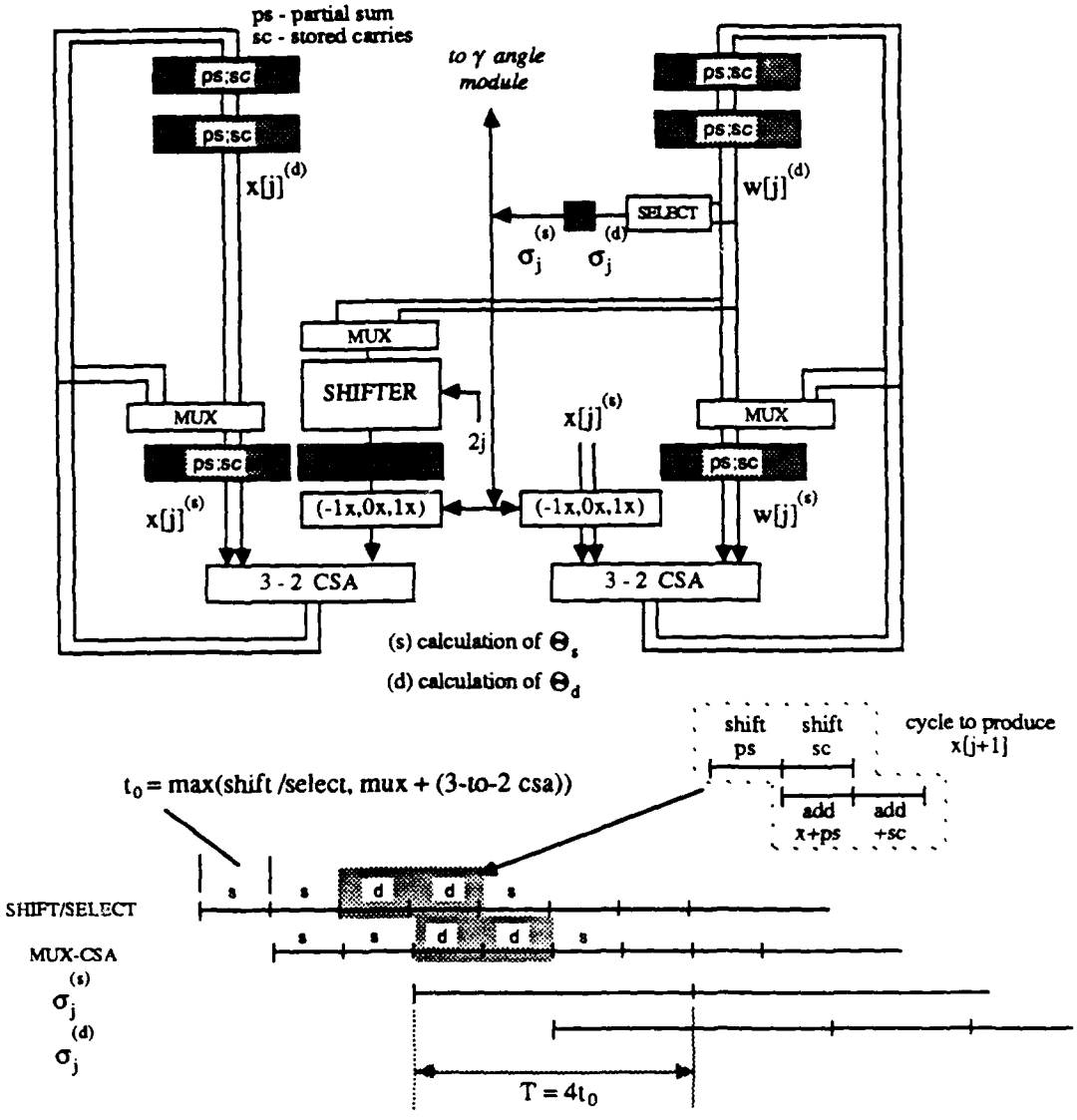

Figure 5. Pipelined Implementation for Θ_s, Θ_d Calculation

We now describe the computation of the γ_j^l ; the computation of γ_j^r being identical (we will omit the superscript to simplify the notation). We perform the computation on-line [ERCE84, IRWI87], to overlap it with the computation of the angles θ_s and θ_d, and with the rotation. To do this, we define the residual

$$z[j] = 2^j \left(\sum_{i=0}^{j+p} \sigma_i \cdot \tan^{-1}(2^{-i}) - \sum_{i=0}^{j} \gamma_i \cdot \tan^{-1}(2^{-i}) \right)$$

where p is the on-line delay. This results in the recurrence,

$$z[j+1] = 2\left(z[j] + \sigma_{j+p} \cdot 2^j \tan^{-1}(2^{-(j+p)}) - \gamma_j \cdot 2^j \tan^{-1}(2^{-j})\right)$$

with initial condition $z[0] = \sum_{i=0}^{p-1} \sigma_i \cdot 2^i \tan^{-1}(2^{-i})$

To simplify the selection function, we decompose this recurrence into two by defining

$$w[j] = z[j] + \sigma_{j+p} \cdot 2^j \tan^{-1}(2^{-(j+p)})$$

so that

$$z[j+1] = 2(w[j] - \gamma_j \cdot 2^j \tan^{-1}(2^{-j}))$$

Note that the multiplication by 2^j is not achieved by shifting, rather the constants $2^j \tan^{-1}(2^{-j})$ are stored in the ROM (instead of $\tan^{-1}(2^{-j})$).

To use carry-save adders for these additions, it is necessary to perform the selection of γ using an estimate of w. The derivation of the selection function is given in [ERCE87d]. The function is

$$\gamma_j = \begin{cases} 1 & \text{if } \hat{w}[j] \geq 1/2 \\ 0 & \text{if } -1/2 \leq \hat{w}[j] \leq 1/4 \\ -1 & \text{if } \hat{w}[j] \leq -3/4 \end{cases}$$

The implementation of this module and its timing is shown in Figure 6. To calculate both angles, the unit is pipelined.

5. Two-sided rotation

The two-sided rotation is done by the sequence of two circular CORDIC operations. We describe the left rotation (the right one is similar); for simplicity, we drop the subscript l. The rotation of the vector M by the angle θ is defined by

$$R[\theta]\begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \quad \textit{where} \quad R[\theta] = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

If the angle is known in its decomposed form, such that $\theta = \sum_{j=0}^{n-1} \gamma_j \tan^{-1}(2^{-j})$ the rotation can be performed by a (partial) CORDIC operation, consisting of the recurrences

$$x[j+1] = x[j] + \gamma_j 2^{-j} y[j] \quad y[j+1] = y[j] - \gamma_j 2^{-j} x[j]$$

with the initial conditions $x[0] = m_1 \quad y[0] = m_2$

After n steps, the result is

$$\begin{bmatrix} x[n] \\ y[n] \end{bmatrix} = KR[\theta]\begin{bmatrix} m_1 \\ m_2 \end{bmatrix} \quad \textit{where} \quad K = \prod_{j=0}^{n-1} (1 + |\gamma_j| \cdot 2^{-2j})^{1/2}$$

The CORDIC operation is partial because it uses the angle produced by another CORDIC operation in decomposed form. Consequently, no angle recurrence is needed. Moreover, the γ's are passed in series (most significant first) so that the rotation can be overlapped with the angle calculation.

To keep up with the fast recurrence step obtained in the computation of the angle when carry-save additions are used, the rotation CORDIC has also to use carry-save addition. Since in the 2x2 matrix, two columns have to be rotated, this can be accomplished by pipelining one unit, without overall speed penalty. The corresponding module is shown in Figure 7.

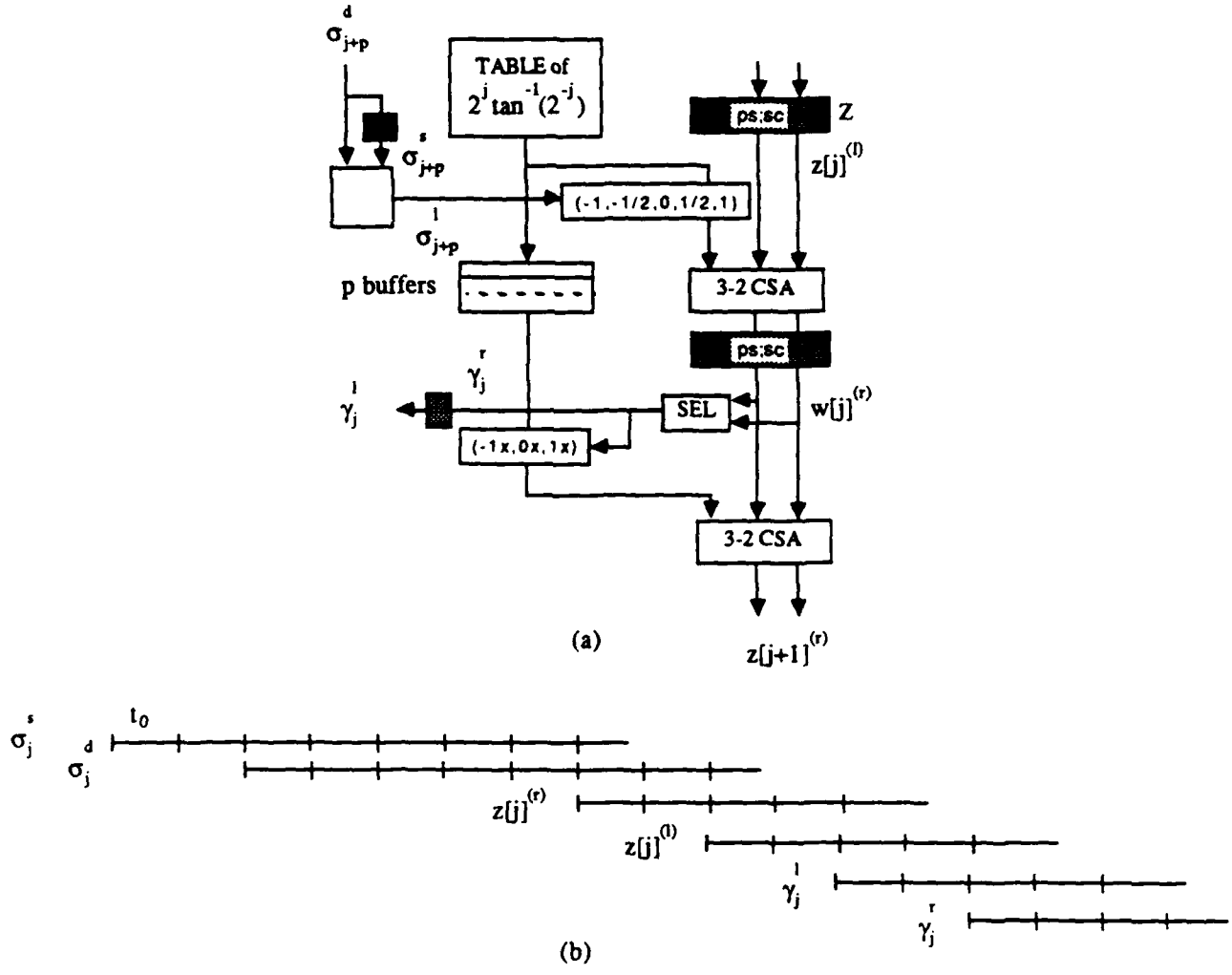

Figure 6: Θ_r, Θ_l Calculation (a) - Implementation, (b) - Timing

For the two-sided rotation, two consecutive rotations are needed. The total time for both rotations is $2n$ cycles. As discussed in Section 2, to match the operations with the way the angles are obtained, it is convenient to interleave the two rotations, so that the j^{th} step of the left rotation is followed by the corresponding step of the right rotation, as shown in the timing diagram of Figure 7. Note that this scheme requires components of each angle to be obtained at a rate of one per four cycles, which is exactly the rate at which they are produced by the angle module.

Floating-point representation

The use of floating-point representation has similar characteristics as for the angle calculation [ERCE87d].

6. Scale-factor correction

Each of the CORDIC rotations produces a modification of the magnitudes [WALT71] by the factor

$$K_l = \prod_{j=0}^{n-1}(1 + (\gamma_j^l)^2 2^{-2j})^{1/2} \quad \text{and} \quad K_r = \prod_{j=0}^{n-1}(1 + (\gamma_j^r)^2 2^{-2j})^{1/2}$$

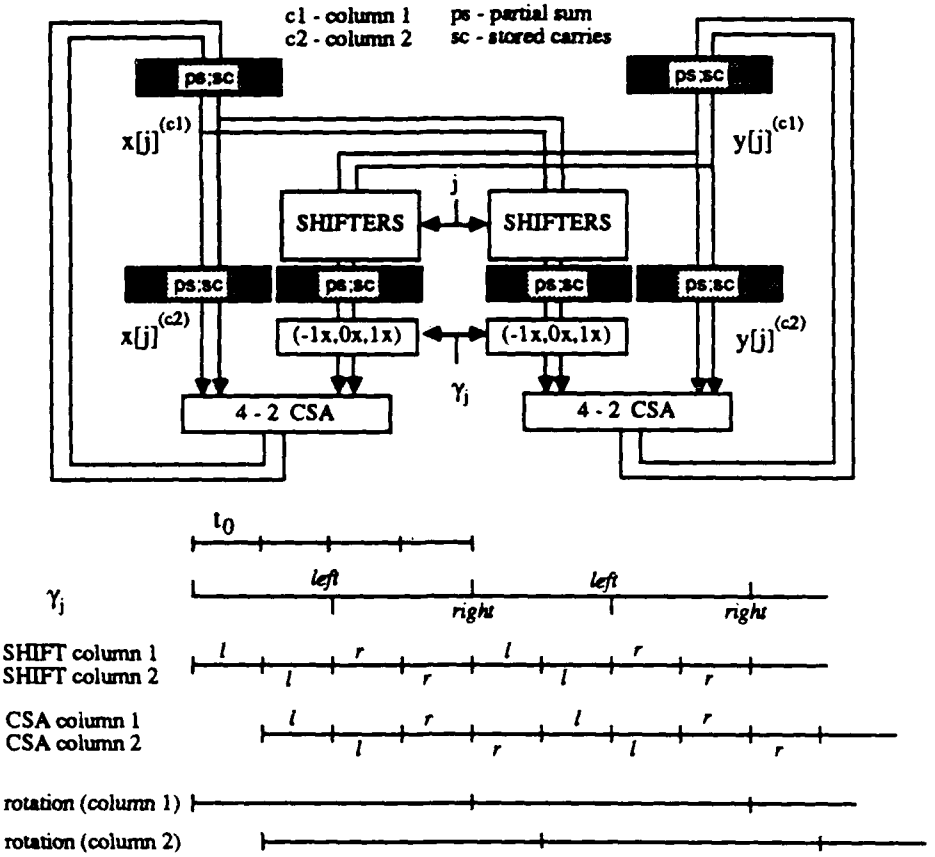

Figure 7. Pipelined Implementationof CORDIC Rotations

It is necessary to correct for these factors. Instead of performing individual corrections, it is possible to perform just one correction using the factor $K = K_l \cdot K_r$. In conventional CORDIC the factors are constant (independent of the actual values of the γ_j's) because the possible values of γ_j is the set {-1,1}. In contrast, in this case the digit sets of θ_l and θ_r are {-1,0,1}, so that the correction factors for each of these rotations are not constant and have to be computed for the specific angle value. Moreover, the correction has to be done by actual division since other methods, such as the one proposed in [DELO83], are valid only if the scaling factor is constant. Recently, Takagi et al. [TAKA87] presented a method for performing the calculation of the angle and the rotation using a redundant adder and, at the same time, having a constant scaling factor. However, their method complicates significantly the recurrences and, therefore, makes the implementation more costly and slower. As a consequence, we prefer to use a variable correction factor.

From the definition,

$$K = K_l \cdot K_r = \prod_{j=0}^{n-1} (1 + (\gamma_j^l)^2 2^{-2j})^{1/2} \cdot \prod_{j=0}^{n-1} (1 + (\gamma_j^r)^2 2^{-2j})^{1/2}$$

The factors K_l and K_r are computed in the angle module and send in an on-line fashion (digit-serial, most-significant digit first) to the rotation modules. There, an on-line multiplication is performed, followed by the (on-line) divisions for correction. These on-line modules are described in [TRIV77]. As part of the division, an on-the-fly conversion [ERCE87c] converts the result to conventional two's complement representation.

We now describe the computation of K_r (the calculation of K_l is similar). The algorithm has two steps:

i) Compute

$$P = \prod_{j=0}^{n-1}(1 + |\gamma_j|2^{-2j})$$

by the recurrence $P[j+1] = P[j] + |\gamma_j| \cdot 2^{-2j} P[j]$ with $P[0] = 1$ and $P = P[n] = P[n/2]$. This recurrence has the same form as the x recurrence in the CORDIC module for angle calculation (Section 3). A module for its computation is shown in Figure 8. Note that only $n/2$ steps are needed, which is also true for the mentioned recurrence x. Consequently, the same module can be used. Moreover, since two factors have to be computed, the unit can be pipelined in the same way as the angle module.

ii) Compute $K_l = P_l^{1/2}$ by a square-root unit. A possible implementation is described in [HWAN78].

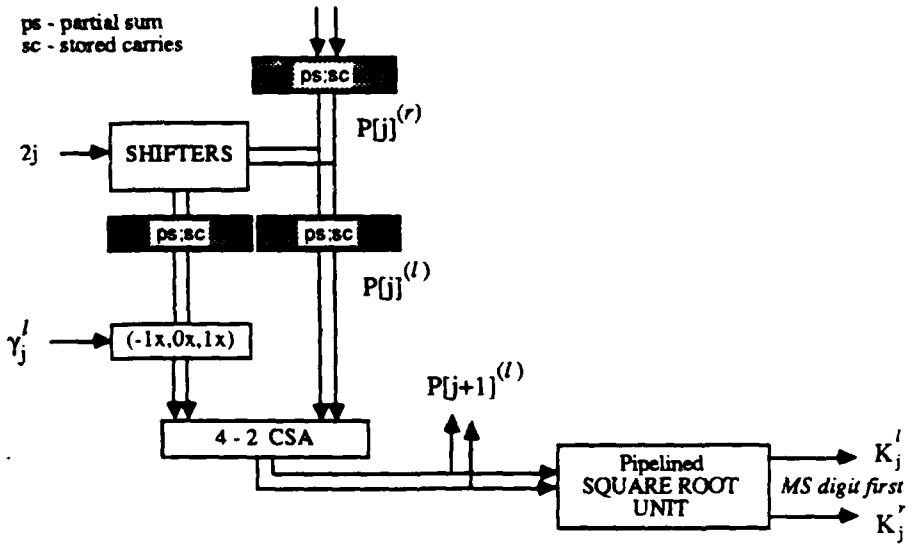

Figure 8. Pipelined Implementation of K^l, K^r calculation

7. Summary of overall SVD system

We now summarize the complete system. The diagonal processors contain the following components:

- A partial redundant CORDIC module to evaluate the angles θ_s and θ_d (in decomposed form). The main components of this module are a carry-save adder and a variable shifter. The module is pipelined with two stages, to compute both angles. This module also computes P_l and P_r.

- An on-line module to compute the decomposition digits γ_j^l and γ_j^r of the angles θ_l and θ_r. The main components of this module are two 3-2 carry-save adders and a ROM to store the angle constants. The module is also pipelined to compute both angles.

- One partial CORDIC module to perform the rotations. Again, this module do not require an angle recurrence, since the angle is produced in suitable decomposed form. The main components of this module are two 4-2 carry-save adders and two double shifters. The module is pipelined to compute the rotations of both columns. Moreover, the rotations are interleaved to match with the way the angles are produced.

- An on-line multiplication module and two on-line division modules to perform the scaling correction. These dividers also convert to conventional representation.

The off-diagonal processors contain the same rotation module as the diagonal processor, one multiplier, and four division modules.

An important property of this implementation is that the communication between modules is digit-serial, which significantly reduces the required connections.

Figure 9 shows the timing of the complete system. We estimate that the operation takes

$$T = 5n + 18 \quad \textit{clock cycles}$$

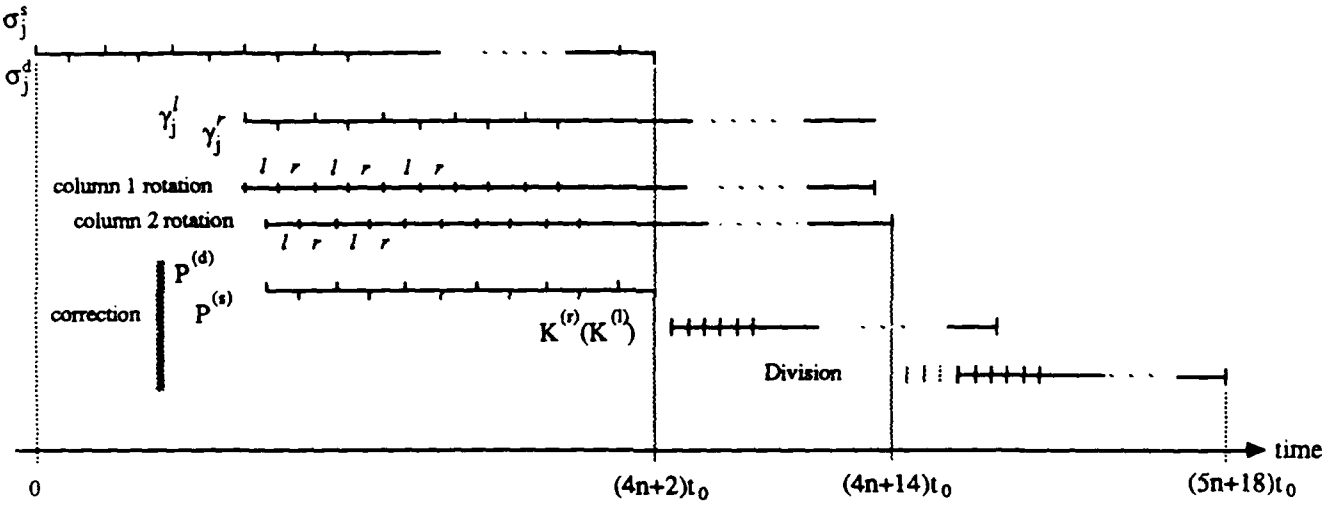

Figure 9. Overall Timing of SVD

As mentioned in Section 2, these clock cycles are short (approximately one 4-2 carry-save adder) which requires a fast clock. If this clock is not suitable, it is possible to replace pipelining by replication; this would achieve the same speed with additional hardware. Moreover, if this still results in a clock that is too fast, several steps of the recurrences can be unfolded; again this would result in the same overall speed with an increase in hardware.

We now compare the speed with that obtained by using conventional CORDIC modules, as described in [CAVA87]. If a pipelined implementation is used also in this case (the comparison would be similar using in both cases a non-pipelined implementation), the number of cycles is $6.5nd$, where d is the delay in basic cycles of a stage, that is, the delay of one half of the sum of the delay of a carry-propagate adder of n bits plus a variable shifter. For the estimated value of d=3, the implementation proposed here is about 3.8 times faster. Moreover, it is 1.7 times faster than the nonredundant scheme proposed in [ERCE87b].

With respect to area, we cannot make a significant comparison without actual realization.

References

[AHME82] H.M. Ahmed, J.M. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," IEEE Computer, Vol.15, No. 1, Jan. 1982, pp. 65-82.

[ATKI75] D. E. Atkins, "Introduction to the Role of Redundancy in Computer Arithmetic," Computer, June 1975, pp.74-75.

[AVIZ61] A. Avizienis, "Signed-Digit Number Representations for Fast Parallel Arithmetic," IEEE Trans. Elec. Computers, Vo. EC-10, September 1961, pp.389-400.

[BREN85a] R.P. Brent, F.T. Luk, and C.F. Van Loan, "Computation of the Singular Value Decomposition Using Mesh-Connected Processors," Journal of VLSI and Computer Systems, vol. 1, no. 3, pp.242-270, 1985.

[BREN85b] R.P. Brent and F.T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," SIAM J. Sci. Statist. Comput., vol. 6, pp. 69-84, 1985.

[CAVA87] J.R. Cavallaro and F.T. Luk, "CORDIC Arithmetic for an SVD Processor," Proc. 8th Symposium on Computer Arithmetic, pp. 113-120, 1987.

[CIMI81] L. Ciminiera, A. Serra, and A. Valenzano, "Fast and Accurate Matrix Triangularization using an Iterative Array," Proceedings 5th. Symposium on Computer Arithmetic, 1981, pp. 215-221

[DELO83] J.M. Delosme, "VLSI Implementation of Rotations in Pseudo-Euclidean Spaces," Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing, 2, pp. 927-930.

[DELO87] J.M. Delosme, "A Processor for Two-Dimensional Symmetric Eigenvalue and Singular Value Arrays," Proc. 21st Asilomar Conference on Signals, Systems, and Computers, 1987, pp. 217-221.

[DEPR84] Ed.F. Deprettere, P. Dewilde, and R. Udo, "Pipelined CORDIC Architectures for Fast Filtering and Array Processing," 1984 IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 41A.61-64, 1984.

[ERCE84] M.D. Ercegovac, "On-Line Arithmetic: An Overview," Proc. SPIE Real-Time Signal Processing, 495 (VII), pp. 86-93, August 1984.

[ERCE87a] M.D. Ercegovac and T. Lang, "On-Line Scheme for computing Rotation Factors," Proc. 8th Symposium on Computer Arithmetic, pp. 196-203, 1987.

[ERCE87b] M.D. Ercegovac and T. Lang, "On-Line Schemes for Computing Rotation Factors for SVD," Proc. SPIE 826, August 1987.

[ERCE87c] M.D. Ercegovac and T. Lang, "On-the Fly Conversion of Redundant into Conventional Representations," IEEE Transactions on Computers, vol. C-36, pp. 895-897, July 1987.

[ERCE87d] M.D. Ercegovac and T. Lang, "Redundant and On-line CORDIC: Application to Matrix Triangularization and SVD," UCLA Computer Science Dept., Report CSD-870046, Sept. 1987 (to be published in IEEE Trans. on Computers).

[GENT81] W.M. Gentleman and H.T. Kung, "Matrix Triangularization by Systolic Arrays," Proc. SPIE: Real-Time Signal Processing IV (1981), pp. 19-26.

[GOLU83] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The John Hopkins University Press, Baltimore, 1983.

[HWAN78] K. Hwang, *Computer Arithmetic*, John Wiley & Sons, 1978.

[IRWI87] M.J. Irwin and R.M. Owens, "Digit-Pipelined Arithmetic as Illustrated by the Paste-Up System: A Tutorial," IEEE Computer, April 1987, pp. 61-73.

[LUK86] F.T. Luk, "Architectures for Computing Eigenvalues and SVDs," Proc. SPIE Highly Parallel Signal Processing Architectures, vol. 614, 1986.

[ROBE58] J.E. Robertson, "A New Class of Digital Division Methods," IEEE Trans. Elec. Computers, Vol. EC-7, September 1958, pp.218-222.

[TAKA87] N. Takagi, T.Asada, S. Yajima, "An Algorithm for Computing Sine and Cosine Using Redundant Binary Representation", Syst. Comput. Japan (USA), vol.18, no.8, p1-9, (August 1987).

[TRIV77] K.S. Trivedi and M.D. Ercegovac, "On-Line Algorithms for Division and Multiplication," IEEE Trans. on Computers Vol. C-26(7), pp.681-687 (July 1977).

[VOLD59] J. Volder, "The CORDIC Trigonometric Computing Technique," IRE Trans. Electronic Computers, EC-8, no. 3, pp. 330-334, Sept. 1959.

[WALT71] J.S. Walther, "A Unified Algorithm for Elementary Functions," AFIPS Spring Joint Computer Conf., pp. 379-385, 1971.

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 5, 209–227 (1988)

On-Line Scheme for Computing Rotation Factors*

MILOŠ D. ERCEGOVAC AND TOMAS LANG

UCLA Computer Science Department, University of California, Los Angeles, California 90024

Received August 19, 1987

An integrated radix-2 on-line algorithm for computing rotation factors for matrix transformations is presented. The inputs are in sign-and-magnitude, floating-point representation and the outputs can be used in on-line signed-digit or in parallel form. The exponents are computed using conventional arithmetic while the significands are processed using on-line algorithms. The conventional result is obtained by using an on-the-fly conversion scheme. The rotation factors are computed in $10 + n$ clock cycles for n-bit significands. The clock period is kept small by the use of redundant adder schemes and low-precision estimates. The implementation and performance of the algorithm are discussed and compared with other approaches.

1. INTRODUCTION

The rotation factors that we propose to compute are an essential part of some matrix transformations such as matrix triangularization methods [7] and the QR decomposition [9]. In particular, in systolic arrays in which the rotations are done in parallel, the time of computation of the rotation factors is critical since it determines the step time of the array. We present an implementation using on-line arithmetic [4, 8, 11, 12], since this approach is potentially advantageous in evaluating arithmetic expressions consisting of sequentially dependent operations. The on-line scheme for the computation of rotation factors was proposed previously in [2], where it is shown that it can produce a significant speed-up with respect to the use of conventional algorithms. However, the description is given at a high level and using previously developed algorithms for the operations of multiplication, division, and square root [3, 10, 12]. In this paper we develop an *integrated* algorithm for the rotation factors, which consists of modifications of the algorithms of

* This research has been supported in part by the ONR under Contract N00014-85-K-0159, "On-Line Arithmetic Algorithms and Structures for VLSI."

0743-7315/88 $3.00

the primitive operations to provide for suitable interfaces with their predecessors and successors in the computation. In this way, both the on-line delay and the complexity of the implementation are reduced. Moreover, the scheme is for floating-point representations.

The clock period is kept small by the use of redundant techniques, such as redundant adders (i.e., carry-save or signed-digit) and selection functions that depend on low-precision estimates. The conversion of the result from redundant to conventional representation is done using an on-the-fly scheme [5]. In principle, it is possible to select for each component operation the most suitable redundant representation. However, preliminary analysis has indicated that there are no clear differences between the use of carry-save and signed-digit approaches. Therefore, we illustrate the scheme by an implementation with carry-save adders throughout. This has the advantage of making the bit slices of all component operations the same.

We propose to compute the rotation factors of the Givens matrix transformation [7], defined as

$$C = \frac{G}{(G^2 + H^2)^{1/2}}, \qquad S = \frac{H}{(G^2 + H^2)^{1/2}}. \tag{1.1}$$

We assume that $G = g \cdot 2^{e_G}$, $H = h \cdot 2^{e_H}$, $C = c \cdot 2^{e_C}$, and $S = s \cdot 2^{e_S}$ are normalized floating-point numbers with n-bit fractions in sign-and-magnitude. In order to reduce the on-line delay, the algorithm for division uses the dividend in bit-parallel form. Since division begins after several clock periods, it should be possible to load operands byte-serially. If this is not acceptable, the division algorithm can be easily modified to accept the dividend in on-line form. The results are produced in both the bit-parallel and the on-line forms. The exponents are represented and processed in a conventional, bit-parallel manner. The special cases $G = 0$ (producing $C = 0$ and $S = 1$) and $H = 0$ (producing $C = 1$ and $S = 0$) are omitted from further discussion.

The scheme, shown in Fig. 1a, performs the following functions:

1. Computation of exponents
2. Computation of aligned fractions $x = align(g)$ and $y = align(h)$
3. Computation of $z = x^2 + y^2$ $(0.25 \leq z < 2)$
4. Computation of $d = z^{1/2}$ $(0.5 \leq d < 2^{1/2})$
5. Computation of $c = g/d$ and $s = h/d$ $(0.5 \leq c, s < 1)$.

Figure 1b shows the timing diagram indicating the on-line delays to be determined later. We now describe each component separately and then comment on the whole system.

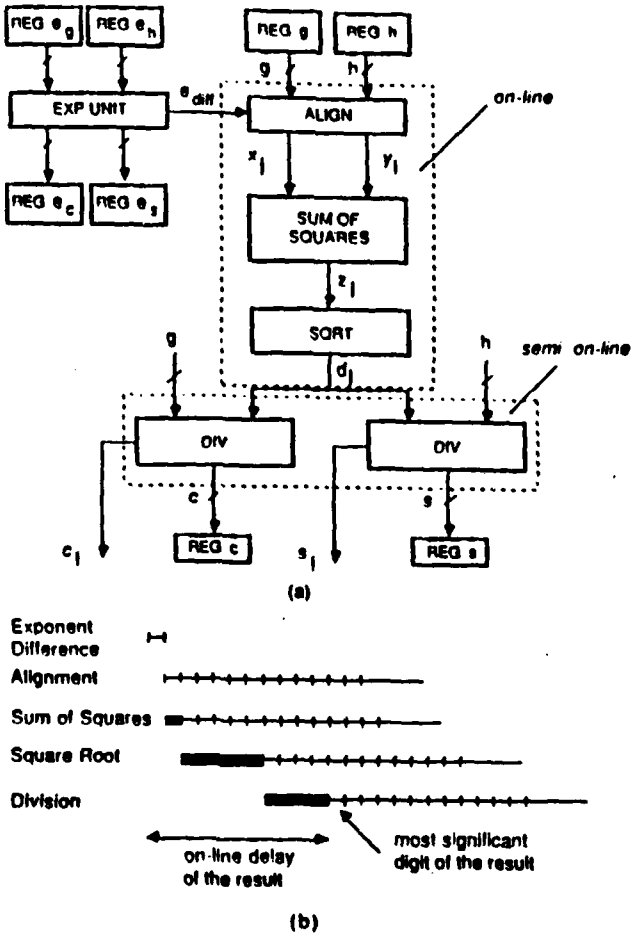

FIG. 1. (a) General scheme. (b) Overall timing.

2. PROCESSING OF EXPONENTS

The exponents of the result can be obtained directly, instead of going through the intermediate steps. Since the sequence of operations is sum of squares, square root, and division, we get *exponent of sum of squares* = $2e$ (where $e = \max(e_G, e_H)$) and *exponent of square root* = e. Therefore, $e_C = e_G - e$ and $e_S = e_H - e$. By defining $e_{diff} = e_G - e_H$ we finally get

$$e_C = \begin{cases} 0 & \text{if } e_{diff} \geq 0 \\ e_{diff} & \text{otherwise} \end{cases} \qquad e_s = \begin{cases} -e_{diff} & \text{if } e_{diff} \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

3. ALIGNMENT

The alignment of the input fractions is performed in an on-line manner during the parallel-to-serial conversion according to following algorithm:

Algorithm *Align*

```
/* Delay operand with smaller exponent */
for i = 1,2,. . .,n
        if e_diff ≥ 0 then
                {x_i = g_i;
                if i ≤ e_diff  then y_i = 0;
                else y_i = h_(i-|e_diff|)}
        else
                {y_i = h_i;
                if i ≤ |e_diff|  then x_i = 0;
                else x_i = g_(i-|e_diff|)}
end Align
```

The alignment scheme is shown in Fig. 2. The exponent difference is obtained in one clock cycle. Thereafter, the bits of the aligned operands are obtained one per cycle. The on-line delay $p_{align} = 1$ due to the calculation of exponent difference.

4. On-Line Sum of Squares Algorithm

In this section we present an algorithm and implementation for on-line sum of squares that integrates well in the computation of the rotation factors.

To incorporate a possible on-line delay p we compute

$$z^* = 2^{-p}z = 2^{-p}(x^2 + y^2). \tag{4.1}$$

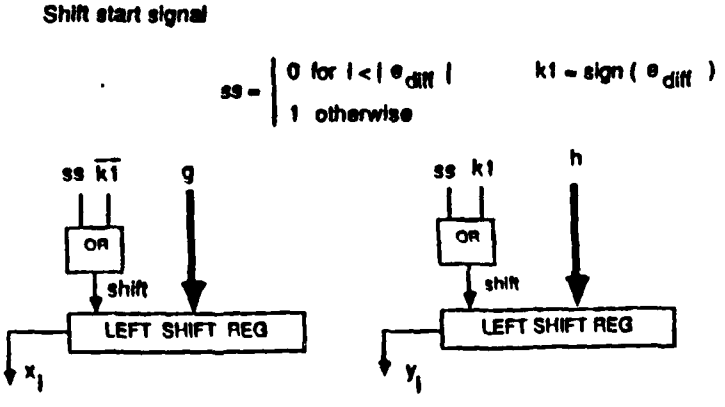

FIG. 2. Alignment scheme.

For the on-line forms of x, y, and z^*,

$$\begin{aligned} X[j] &= X[j-1] + x_j 2^{-j}, & X[0] &= 0 \\ Y[j] &= Y[j-1] + y_j 2^{-j}, & Y[0] &= 0 \\ Z[j] &= Z[j-1] + z_j^* 2^{-j}, & Z[0] &= 0, \end{aligned} \tag{4.2}$$

we define the residual function

$$W[j] = 2^j(2^{-p}X[j]^2 + 2^{-p}Y[j]^2 - Z[j]). \tag{4.3}$$

To obtain the correct result, the residual is bounded so that $W[j] < 1$ which results in

$$(2^{-p}X[n]^2 + 2^{-p}Y[n]^2 - Z[n]) < 2^{-n}. \tag{4.4}$$

From the residual expression we get the recurrence

$$\begin{aligned} W[j] = 2W[j-1] &+ (2X[j-1]x_j + x_j^2 2^{-j})2^{-p} \\ &+ (2Y[j-1]y_j + y_j^2 2^{-j})2^{-p} - z_j^* \end{aligned} \tag{4.5}$$

with the initial condition

$$W[0] = (X[0]^2 + Y[0]^2)2^{-p} - Z[0] = 0. \tag{4.6}$$

To keep $W[j]$ bounded we make the selection

$$\begin{aligned} z_j^* = \mathit{integer}\{2W[j-1] &+ (2X[j-1]x_j + x_j^2 2^{-j})2^{-p} \\ &+ (2Y[j-1]y_j + y_j^2 2^{-j})2^{-p}\} \end{aligned} \tag{4.7}$$

which transforms the recurrence into

$$\begin{aligned} W[j] = \mathit{fraction}\{2W[j-1] &+ (2X[j-1]x_j + x_j^2 2^{-j})2^{-p} \\ &+ (2Y[j-1]y_j + y_j^2 2^{-j})2^{-p}\}. \end{aligned} \tag{4.8}$$

To have a short step time, this recurrence is implemented using a redundant addition (e.g., carry-save, signed-digit). As mentioned in the Introduction, here we use the carry-save adder variant. In this case, the exact integer and fraction parts cannot be determined (without propagating carries). Consequently, the previous expressions are transformed into

$$z_j^* = csint\{2W[j-1] + (2X[j-1]x_j + x_j^2 2^{-j})2^{-p} + (2Y[j-1]y_j + y_j^2 2^{-j})2^{-p}\} \quad (4.9)$$

$$W[j] = csfrac\{2W[j-1] + (2X[j-1]x_j + x_j^2 2^{-j})2^{-p} + (2Y[j-1]y_j + y_j^2 2^{-j})2^{-p}\}, \quad (4.10)$$

where *csfrac* produces a value in the range [0, 2).

The corresponding carry-save operation is shown in Fig. 3a. From it we see that the bounds on z_j^* are (for $x, y < 1$)

$$0 \leq z_j^* \leq 6 \quad \text{if} \quad p = 0, \quad 0 \leq z_j^* \leq 4 \quad \text{if} \quad p = 1. \quad (4.11)$$

These values of z_j^* are over-redundant (that is, they are larger than 1 for a radix-2 representation). We choose this alternative because it simplifies the selection, reduces the on-line delay, and is suitable for the interface with the square root. We choose to implement the case with $p = 0$ to minimize the overall on-line delay. The result is in the range $0.25 \leq z < 2$. The algorithm is summarized next.

Algorithm *Sum of Squares*

```
/* Initialization */
W[0] ← 0; z_0 ← 0;
X[0] ← 0; Y[0] ← 0;

/* Recurrence */
for j = 1,2,...,n+1
   {W[j] ← csfract{2W[j-1] + (2X[j-1]+x_j 2^-j)x_j
                           + (2Y[j-1]+y_j 2^-j)y_j};
   z_j ← csint{2W[j-1] + (2X[j-1]+x_j 2^-j)x_j
                           + (2Y[j-1]+y_j 2^-j)y_j};
   X[j] ← append(X[j-1], x_j);
   Y[j] ← append(Y[j-1], y_j)}
```

end *Sum of Squares*

Implementation

The scheme is shown in Fig. 3b. Note that its implementation complexity is similar to that of an on-line multiplier. The on-line delay is $p_{ss} = 0$. The critical path consists of two multiplexers, one 4-to-2 carry-save adder, and a 3-bit carry-propagate adder.

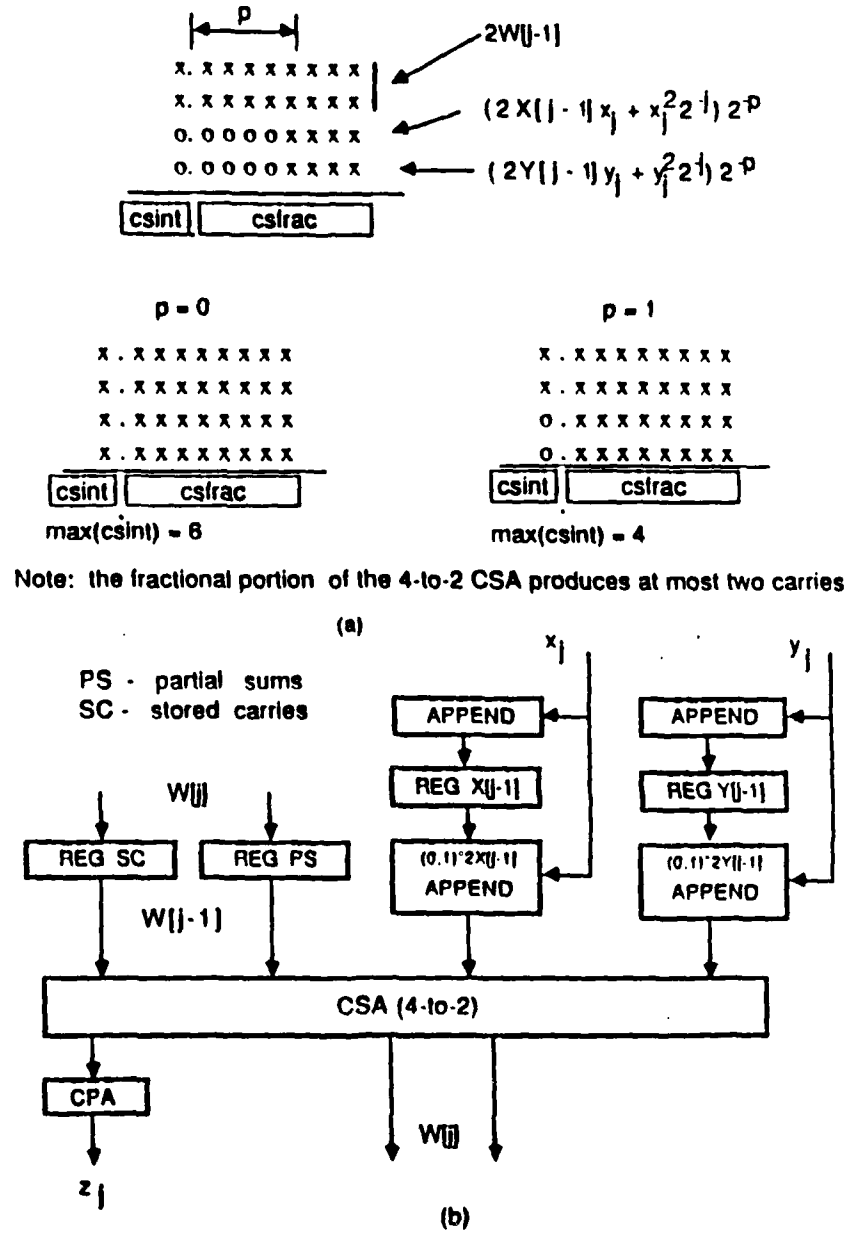

FIG. 3. (a) Carry-save operation in sum of squares. (b) Scheme.

5. ON-LINE SQUARE ROOT

We now describe an on-line square root algorithm to interface with the sum of squares and the division, for the computation of rotation factors. This algorithm allows an over-redundant input digit set, differing in this respect from previous on-line square root algorithms [3, 10].

Let the on-line forms of the argument z and of the square root d be

$$Z[j] = Z[j-1] + z_j 2^{-j}, \qquad -b \leq z_j \leq a$$
$$D[j] = D[j-1] + d_j 2^{-j}, \qquad d_j \in \{-1, 0, 1\}. \tag{5.1}$$

To bound the error of the computation we make

$$(D[j] - 2^{-j})^2 + b2^{-p-j} \leq Z[j+p] \leq (D[j] + 2^{-j})^2 - a2^{-p-j}, \tag{5.2}$$

where p is the on-line delay determined in Appendix A and, as indicated, the argument digits satisfy $z_j \in \{-b, \ldots, 0, \ldots, a\}$.

We define a residual $R[j]$ such that

$$R[j] = 2^j(Z[j+p] - D[j]^2), \qquad R[0] = Z[p]. \tag{5.3}$$

From (4.2) the residual should satisfy the bounds $C_{-1}[j] \leq R[j] \leq C_1[j]$ where

$$C_{-1}[j] = (-2D[j] + 2^{-j} + b2^{-p}), \qquad C_1[j] = (2D[j] + 2^{-j} - a2^{-p}). \tag{5.4}$$

The resulting recurrence is

$$R[j] = 2R[j-1] + z_{j+p}2^{-p} - 2d_j D[j-1] - d_j^2 2^{-j}. \tag{5.5}$$

As stated before, to have a short step time, we use a redundant addition and a selection function that depends on a low-precision estimate. The corresponding selection function *dsel* is derived in Appendix A. Using a carry-save adder, $t = 3$ fractional bits in the estimate of the remainder, and the on-line delay $p_{sqr} = 4$, we obtain the selection function

$$d_j = dsel(\hat{R}^*[j-1]) = \begin{cases} 1 & \text{if} \quad \hat{R}^*[j-1] \geq 0 \\ 0 & \text{if} \quad \hat{R}^*[j-1] = -\frac{1}{2} \\ -1 & \text{if} \quad \hat{R}^*[j-1] \leq -\frac{1}{2}, \end{cases}$$

where $\hat{R}^*[j-1]$ is an estimate with three fractional bits of $R^*[j-1] = R[j-1] + z_{j+p}2^{-p-1}$.

The corresponding algorithm for on-line square root is summarized next.

Algorithm *Sqrt*

/* Initialization */

$R[-5] \leftarrow 0$; $D[-1] \leftarrow 0$;

/* Accumulate 4 input digits */

for $j = -4,-3,-2,-1$

$\{R[j] \leftarrow 2R[j-1] + z_{4+j}2^{-4}\}$

/*Recurrence*/

for $j = 0,1,2,\ldots,n$

$\{\hat{R}^*[j-1] = \hat{R}[j-1] + z_{4+j}2^{-5}$;
$d_j = dsel(\hat{R}^*[j-1])$;
$R[j] \leftarrow 2R[j-1] + z_{4+j}2^{-4} - 2d_jD[j-1] - d_j^2 2^{-j}$;
$D[j] \leftarrow convert(D[j-1], d_j)\}$

end *Sqrt*

Note that $z_k = 0$ for $k > n$. The result is in the range $[\frac{1}{2}, 2^{1/2})$.

Implementation

The square root scheme is shown in Fig. 4a. The internal arithmetic is performed in 2's complement system. The signed-digit operand is converted to 2's complement using the on-the-fly conversion method [5]. The operation performed by APPEND* module is discussed in Appendix C. The critical path (Fig. 4b), consists of a 5-bit CPA, one multiplexer, and a 3-to-2 carry-save adder.

6. On-Line Division Algorithm and Implementation

We now present the algorithm for the division operation. We consider the computation of s (that of c is similar). To incorporate the necessary on-line delay p we redefine the division operation as

$$s = 2^{-p}\frac{h}{d}, \qquad h \in \left[\frac{1}{2}, 1\right), \qquad d \in \left[\frac{1}{2}, 2^{1/2}\right), \tag{6.1}$$

where s is obtained without delay. The real quotient is obtained by shifting left s by p positions. Consequently, the real quotient is obtained with an on-line delay of p.

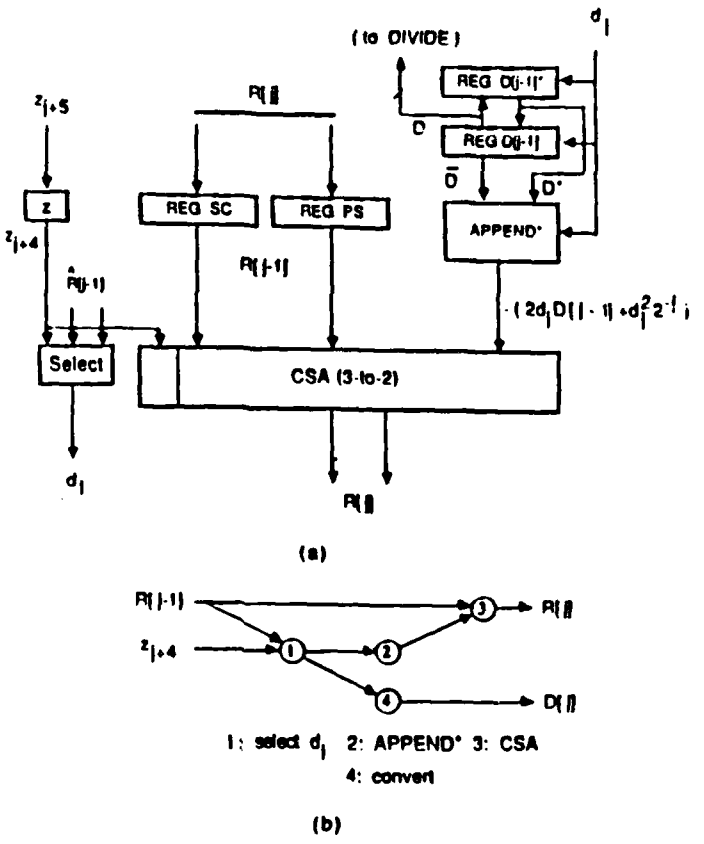

FIG. 4. (a) Square root scheme. (b) Critical paths.

Since we are developing an algorithm that is specifically suited to the computation of the rotation factors, we assume that h (the dividend) is known in parallel form (not on-line), while d is on-line and s is also produced on-line (but converted on-the-fly to conventional representation). However, this approach requires that the dividend be communicated to the division unit in parallel. Because of this, it might be more convenient for a particular implementation to use an algorithm which also uses the dividend in an online fashion. This would require minor modifications to the algorithm. The formulation using the dividend off-line leads to the following recurrence. Let

$$P[j] = 2^j(2^{-p}h - S[j] \times D[j]) \tag{6.2}$$

be a partial remainder, satisfying the bound $|P[j]| < |D[j]|$.

The corresponding recurrence is

$$P[j] = 2P[j-1] - S[j-1]d_j - D[j]s_j, \qquad P[0] = 2^{-p}h, \qquad S[0] = 0. \tag{6.3}$$

The recurrence can be decomposed into the two steps

$$W[j] = 2P[j-1] - S[j-1]d_j \tag{6.4}$$

$$P[j] = W[j] - D[j]s_j \tag{6.5}$$

so that the selection function can depend on W.

As for the other algorithms, to have a short clock period, a redundant adder is used, and the selection is done on a low-precision estimate of W. The selection function *qsel* is determined in Appendix B. For an estimate of $t = 3$ fractional bits and an on-line delay of $p_{div} = 3$, and using carry-save adder, the selection function is

$$s_j = qsel(\hat{W}[j]) = \begin{cases} 1 & \text{if} \quad \hat{W}[j] \geq \frac{1}{8} \\ 0 & \text{if} \quad -\frac{1}{8} \leq \hat{W}[j] \leq \frac{1}{8} \\ -1 & \text{if} \quad \hat{W}[j] \leq -\frac{1}{8}. \end{cases}$$

The corresponding algorithm is given next.

Algorithm *Division*

/* Initialization */
$P[0] \leftarrow h \times 2^{-3}$; $D[0] \leftarrow d_0$; $S[0] \leftarrow 0$; $s_0 \leftarrow 0$;

/* Recurrence */

for $j = 1,2,\ldots,n+3$:

{ $W[j] = 2P[j-1] - S[j-1]d_j$;

/* Note that $S[j-1] < 2^{-p} = 2^{-3}$ */

$\hat{W}[j] = (2P[j-1])_4 - (1 - sign(d_j)) * 2^{-3}$;

$s_j = qsel(\hat{W}[j])$;
$P[j] \leftarrow W[j] - D[j]s_j$;
$S[j] \leftarrow convert\ (S[j-1], s_j)$ }

end *Division*

Note that $(A)_4$ denotes the four most significant bits of A and $d_k = 0$ for $k > n$.

Implementation

The implementation is shown in Fig. 5a. Note that the additions/subtractions are done using carry-save adders in 2's complement system; for this

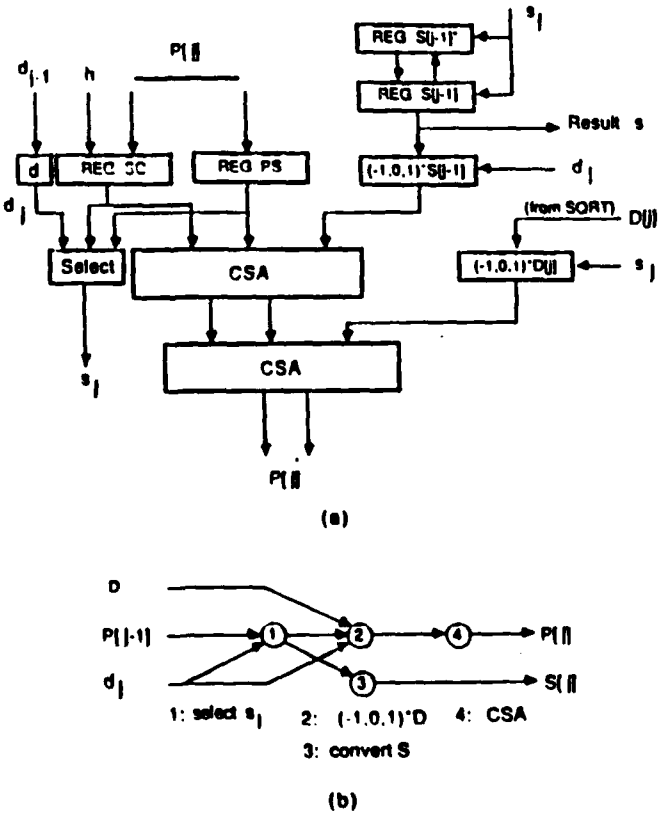

FIG. 5. (a) Division scheme. (b) Critical paths.

reason, the quotient is converted on-the-fly [5] into conventional form to be used as operands for the additions. The critical path is shown in Fig. 5b. We estimate that its delay corresponds to two multiplexers, one 4-bit CPA and one CSA.

7. SUMMARY OF THE PROPOSED SCHEME AND COMPARISONS

An integrated scheme for computing rotation factors using on-line arithmetic algorithms has been presented. The scheme has the following characteristics:

–The inputs and outputs are in sign-and-magnitude floating-point format; the inputs may be loaded byte-serially if appropriate. The outputs can also be used in on-line signed-digit form.

–The on-line delay of the scheme, defined as the number of clock cycles between loading of the operands and the appearance of the most significant digit of the result, is

$$P_{rot} = (p_{align} + 1) + (p_{ss} + 1) + (p_{sqr} + 1) + (p_{div} + 1)$$
$$= 1 + 1 + 5 + 4 = 11.$$

–The latency, defined as the total time to generate the result, is $T = p_{rot} + n - 1 = 10 + n$ for n-bit significands (Fig. 1b).

–The cycle time is determined roughly by a multiplexer, a 4-to-2 carry-save adder, and a 5-bit carry-propagate adder.

–Since it takes roughly k cycles to propagate the kth bit of an on-line unit to influence the selection function, the number of bit slices required to implement the carry-save adders is approximately $n/2$.

Comparison with Alternative Schemes

We now compare the execution time of the presented on-line implementation with that of other schemes proposed for the computation of the rotation factors. We just do a rough comparison since a more accurate one would require many assumptions on the implementation of these other schemes as well as on the technology. To remain independent of the particular technology, we make the comparison in terms of number of clock cycles, instead of using actual time. Even so, we must make some assumptions, since not all clock cycles are the same. Therefore, we define a *basic* clock cycle and relate all others to this basic one. The basic clock cycle has a delay of roughly a multiplexer, a 4-to-2 (carry-save) adder, a selection function composed of a short carry-propagate adder and some logic, and the loading of a register. This basic clock cycle is suitable for the on-line implementation presented in this paper.

As shown in Table I, we compare with the following schemes, assuming 32-bit significands and single-chip implementations:

(a) Using only one floating-point multiplier (array multiplier) and one floating-point adder. For this scheme, the divisions and the square root use an iterative multiplicative approach. We estimate the number of floating-point operations to be about 20 (1 addition + 2 multiplications + 1 square

TABLE I
COMPARISON WITH OTHER SCHEMES

Scheme	Basic cycles	Speed-up
Multiplier + adder	120	1
MULT + ADD + SQR + DIV	$5n = 160$	0.75
2MULT + ADD + SQR + 2DIV	$3n = 96$	1.25
CORDIC	$7.5n = 226$	0.55
Redundant/on-line CORDIC	$2n = 64$	1.9
On-line (this paper)	$10+n = 42$	3

root + 1 reciprocal + 2 multiplications). We estimate that the time for each of these floating-point operations is of 6 basic clock cycles. This results in a total of 120 basic clock cycles.

(b) Using one adder, one multiplier, one divider, and one square root unit. In this case we assume that the multiplier is sequential (to fit all units in one chip). Each unit takes n cycles (except addition, which is neglected) for a total of $5n$ clock cycles. Fast algorithms (with carry-save addition) can be used so that the clock cycle corresponds roughly to the basic cycle.

(c) Using one adder, two multipliers, two dividers, and one square root unit. In this case the parallelism in the computation can be exploited to achieve a time of $3n$ clock cycles.

(d) It is possible to use a CORDIC approach to calculate the rotation angle and also to perform the actual rotation [1]. The operation consists in performing roughly $1.25n$ iterations of the CORDIC recurrence. The clock cycle for this recurrence is larger than the basic clock cycle because it involves a variable shift and a carry-propagate addition. We estimate that the CORDIC cycle is about six times the basic cycle, resulting in a time of $7.5n$ basic cycles.

(e) The speed of the CORDIC recurrence can be increased by using a redundant adder and performing the operation on-line to eliminate the shifter [6]. This results in an execution time of about $2n$ cycles.

The speed-up with respect to case (a) is shown in the table. As can be seen, the scheme presented here is the fastest and produces a speed-up of about 3. The next fastest is the redundant and on-line CORDIC, which might have the advantage of smaller area.

It is difficult to compare the schemes with respect to implementation complexity without doing detailed designs. However, we can make the following general statements:

(i) The general-purpose approach of using one multiplier and one adder is probably inefficient in area since it requires fast modules to be competitive in speed.

(ii) The on-line approach is comparable in complexity with respect to the specialized non-on-line case (c) and provides better performance. Moreover, the on-line approach has the advantage of reduced interconnection among modules. It also uses only about $n/2$ bit slices per operation.

(iii) The CORDIC approach is promising because of the complex operation performed by the single CORDIC module. However, to compete in speed it is necessary to use the redundant on-line version, which complicates somewhat the implementation. Details of this approach are given in [6].

APPENDIX A: SELECTION FUNCTION FOR SQUARE ROOT

We determine here a selection function *dsel* for d_j so that the bounds on the residual $R[j]$ are satisfied. For this, we compute the intervals $[L_k, U_k]$ of $R[j-1]$ so that the value $d_j = k$ ($k = -1, 0, 1$) can be selected and $R[j]$ is inside the allowed interval of bounds defined by (5.4). The expression for $R[j-1]$ is $R[j-1] = 2^{-1}R[j] - z_{j+p}2^{-p-1} + d_jD[j-1] + d_j^2 2^{-j-1}$. The resulting intervals are the following:

for $d_j = 1$,

$$\begin{aligned} L_1 &= -D[j] + 2^{-j-1} + b2^{-p-1} - z_{j+p}2^{-p-1} + D[j-1] + 2^{-j-1} \\ &= b2^{-p-1} - z_{j+p}2^{-p-1} \\ U_1 &= D[j] + 2^{-j-1} - a2^{-p-1} - z_{j+p}2^{-p-1} + D[j-1] + 2^{-j-1} \\ &= 2D[j-1] + 2^{-j+1} - a2^{-p-1} - z_{j+p}2^{-p-1}, \end{aligned}$$

for $d_j = 0$,

$$\begin{aligned} L_0 &= -D[j] + 2^{-j-1} + b2^{-p-1} - z_{j+p}2^{-p-1} \\ &= -D[j-1] + 2^{-j-1} + b2^{-p-1} - z_{j+p}2^{-p-1} \\ U_0 &= D[j-1] + 2^{-j-1} - a2^{-p-1} - z_{j+p}2^{-p-1}, \end{aligned}$$

for $d_j = -1$,

$$\begin{aligned} L_{-1} &= -D[j] + 2^{-j-1} + b2^{-p-1} - z_{j+p}2^{-p-1} - D[j-1] + 2^{-j-1} \\ &= -2D[j-1] + 2^{-j+1} + b2^{-p-1} - z_{j+p}2^{-p-1} \\ U_{-1} &= D[j] + 2^{-j-1} - a2^{-p-1} - z_{j+p}2^{-p-1} - D[j-1] + 2^{-j-1} \\ &= -a2^{-p-1} - z_{j+p}2^{-p-1}. \end{aligned}$$

Since $(-z_{j+p}2^{-p-1})$ appears in all the expressions, we will base the selection on

$$R^*[j-1] = R[j-1] + z_{j+p}2^{-p-1}. \tag{A.1}$$

Consequently,

$$\begin{aligned} L_1^* &= b2^{-p-1}, & U_1^* &= 2D[j-1] + 2^{-j+1} - a2^{-p-1} \\ L_0^* &= -D[j-1] + 2^{-j-1} + b2^{-p-1}, & U_0^* &= D[j-1] + 2^{-j-1} - a2^{-p-1} \\ L_{-1}^* &= -2D[j-1] + 2^{-j+1} + b2^{-p-1}, & U_{-1}^* &= -a2^{-p-1}. \end{aligned}$$

The containment conditions, $U_1 \leq C_1[j-1]$ and $L_{-1} \geq C_{-1}[j-1]$, require that $-b \leq z_j \leq a$ which is satisfied by selecting a and b according to (4.11).

The continuity of the selection intervals and the use of redundant adders require that the interval overlaps satisfy the conditions

$$\Delta_{01} = U_0^* - L_1^* = D[j-1] + 2^{-j-1} - (a+b)2^{-p-1} \geq 2^{-t+1}$$

$$\Delta_{10} = U_{-1}^* - L_0^* = D[j-1] - 2^{-j-1} - (a+b)2^{-p-1} \geq 2^{-t+1}, \quad \text{(A.2)}$$

where t is the precision of the assimilated remainder $\hat{R}$.

Since $z \geq 2^{-2}$ (from the sum of squares), the smallest possible value of $d = 2^{-1}$. Therefore, for $a = 6$ and $b = 0$ (as produced by the sum of squares), we get

$$\Delta_{01}\text{min} = 2^{-1} - 3 \cdot 2^{-p} \geq 2^{-t+1} \qquad \Delta_{10}\text{min} = 2^{-1} - 2^{-j-1} - 3 \cdot 2^{-p} \geq 2^{-t+1}. \quad \text{(A.3)}$$

Since $d \geq \frac{1}{2}$, the first negative digit cannot happen for $j < 3$. Therefore, the positive overlaps imply $p \geq 4$ and $t \geq 3$.

We now determine the selection function for $p = 4$, $a = 6$, $b = 0$, and $t = 3$. The overlap region for the selection of 0 or 1 is bounded by

$$L_1^*(\text{max}) = b2^{-p-1} = 0$$

$$U_0^*(\text{min}) = D[j-1] + 2^{-j-1} - a2^{-p-1} > 2^{-1} - 6 \cdot 2^{-5} = \tfrac{5}{16}. \quad \text{(A.4)}$$

Similarly, the overlap region for choosing between 0 and -1 is

$$L_0^*(\text{max}) = -D[j-1] + 2^{-j-1} + b2^{-p-1} \leq -2^{-1} + 2^{-3-1} + 2^{-4} = -\tfrac{1}{8}$$

$$U_{-1}^*(\text{min}) = -a2^{-p-1} = -\tfrac{3}{16}. \quad \text{(A.5)}$$

Since the error of the estimate is always positive (because of the use of carry-save adders and 2's complement representation) and can be at most $2^{-t+1} = \frac{1}{4}$, we get the selection function

$$d_j = dsel(\hat{R}^*[j-1]) = \begin{cases} 1 & \text{if} \quad \hat{R}^*[j-1] \geq 0 \\ 0 & \text{if} \quad \hat{R}^*[j-1] = -\frac{1}{4} \\ -1 & \text{if} \quad \hat{R}^*[j-1] \leq -\frac{1}{2}. \end{cases}$$

APPENDIX B: QUOTIENT SELECTION FUNCTION

We determine here the selection function *qsel* for the quotient digit s_j. This selection function is such that the remainder is inside the required bounds.

First we calculate the actual bounds on $P[j]$ and then the intervals of $W[j]$ for which a selection of $s_j = k$ can be made.

From the recurrence, since $|S[j]| \leq 2^{-p}$, we get

$$L_k - 2^{-p} - kD[j] \leq P[j] \leq U_k + 2^{-p} - kD[j],$$

where $[L_k, U_k]$ is the interval of $2P[j-1]$ for which it is possible to select $s_j = k$.

The containment of the remainder requires that

$$c_{-1} = \frac{L_{-1}}{2} \leq P[j] \leq \frac{U_1}{2} = c_1.$$

Consequently,

$$U_1 = 2D[j] - 2^{-p+1}, \qquad L_{-1} = -2D[j] + 2^{-p+1}$$

and

$$c_1 = -c_{-1} = c = D[j] - 2^{-p}. \tag{B.1}$$

The selection is based on $W[j]$ defined by

$$W[j] = 2P[j-1] - S[j-1]d_j = P[j] + D[j]s_j. \tag{B.2}$$

We now determine the selection intervals $[A_k, B_k]$ of $W[j]$ such that $s_j = k$ can be selected. From (B.1) and (B.2) we get

$$A_k = -c + kD_j \leq W_j \leq c + kD_j = B_k \tag{B.3}$$

and replacing c, $A_k = (k-1)D[j] + 2^{-p}$ and $B_k = (k+1)D[j] - 2^{-p}$.

The value of p is determined to ensure sufficient overlap between the intervals. Since we want to perform the computation of $W[j]$ using carry-save addition, and base the selection on an estimate $\hat{W}$ produced with an assimilation over t fractional bits, the overlap must be at least 2^{-t+1}. Consequently,

$$\Delta(k, k-1) = B_{k-1} - A_k = D[j] - 2^{-p+1} \geq 2^{-t+1} \tag{B.4}$$

so that

$$2^{-p} \leq \frac{D[j] - 2^{-t+1}}{2}. \tag{B.5}$$

For $D[j] \geq \frac{1}{2}$ we get $2^{-p} \leq (1 - 2^{-t+2})/4$. A possible solution to this inequality is $p = 3$ and $t = 3$.

Using these values we determine the selection intervals and the selection function:

s_j	$\hat{W}[j]$
-1	$[-2D[j] + 2^{-3}, -2^{-3})$
0	$[-D[j] + 2^{-3}, D[j] - 2^{-3})$
1	$[2^{-3}, 2D[j] - 2^{-3})$

To have a selection that is independent of the value of $D[j]$ we look at the values of $D[j]$ that produce the smallest intervals. This occurs in all cases for $D[j] = \frac{1}{2}$. The resulting intervals are

s_j	$\hat{W}[j]$
-1	$[-\frac{7}{8}, -\frac{1}{8})$
0	$[-\frac{3}{8}, +\frac{3}{8})$
1	$[+\frac{1}{8}, +\frac{7}{8})$

Since the use of carry-save addition and 2's complement representation produce $\hat{W} = W - \epsilon$ with $0 \leq \epsilon \leq 2^{-t+1}$, we get as selection function

$$s_j = qsel(\hat{W}[j]) = \begin{cases} 1 & \text{if} \quad \hat{W}[j] \geq \frac{3}{8} \\ 0 & \text{if} \quad -\frac{3}{8} \leq \hat{W}[j] \leq \frac{1}{8} \\ -1 & \text{if} \quad \hat{W}[j] \leq -\frac{5}{8}. \end{cases}$$

APPENDIX C: APPENDING OPERATION FOR SQUARE ROOT ALGORITHM

The operand produced by the selection/appending module APPEND* in the square root scheme is

$$q = -(2d_iD[i-1] + d_i^2 2^{-i}).$$

Consider the three possible values of d_i.

(i) For $d_i = 1$, $q = -(2D[i-1] + 2^{-i})$ which, as a string of bits, is

$$-[D, 0, 1_i, 0, 0, 0 \cdots, 0] \qquad (1_i \text{ to indicate } i\text{th position}).$$

So, the 2's complement is

$$\begin{array}{r} \bar{D}, 1, 0_i, 1, 1, 1, \ldots, 1 \\ \qquad\qquad\qquad\qquad\qquad 1 \\ \hline \bar{D}, 1, 1_i, 0, 0, 0, \ldots, 0, \end{array}$$

where $\bar{D}$ is the bit complement of D. That is, in this case we concatenate $\bar{D}$ with 11.

(ii) For $d_i = -1$, $q = 2D[i-1] - 2^{-i}$ which as a weighted string can be written as

$$D, 0, -1_i, 0, 0, 0, \ldots, 0 = (D-1), 1, 1_i, 0, 0, 0, \ldots, 0.$$

Since $D^* = D - 1$ because of the conversion algorithm, we obtain

$$D^*, 1, 1_i, 0, 0, 0, \ldots, 0.$$

That is, in this case we concatenate D^* with 11.

(iii) For $d_i = 0$, $q = 0$.

ACKNOWLEDGMENTS

We thank Dr. J. G. Nash of Hughes Research Laboratories, Malibu, for his interest and support, and referees for their comments.

REFERENCES

1. Ahmed, H. M., Delosme, J. M., and Morf, M. Highly concurrent computing structures for matrix arithmetic and signal processing. *IEEE Comput.* 15, 1 (Jan. 1982), 65–82.
2. Ciminiera, L., Serra, A., and Valenzano, A. Fast and accurate matrix triangularization using an iterative structure. *Proc. 5th IEEE Symposium on Computer Arithmetic*, Ann Arbor, 1981, pp. 215–221.
3. Ercegovac, M. D. An on-line square root algorithm. *Proc. 4th IEEE Symposium on Computer Arithmetic*, Santa Monica, 1978, pp. 183–189.
4. Ercegovac, M. D. On-line arithmetic: An overview. *Proc. SPIE 1984*, Vol. 495, *Real Time Signal Processing VII*, 1984, pp. 86–93.
5. Ercegovac, M. D., and Lang, T. On-the-fly conversion of redundant into conventional representations. *IEEE Trans. Comput.* (July 1987), 895–897.
6. Ercegovac, M. D., and Lang, T. *Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD*. UCLA Computer Science Department, Tech. Rep. CSD-870046, Sept. 1987.
7. Golub, G. H., and Van Loan, C. F. *Matrix Computations*. The Johns Hopkins Univ. Press, Baltimore, 1983.
8. Irwin, M. J., and Owens, R. M. Digit-pipelined arithmetic as illustrated by the paste-up system: A tutorial. *IEEE Comput.* (Apr. 1987), 61–73.
9. Luk, F. T. Architectures for computing eigenvalues and SVDs. *SPIE Proc. Highly Parallel Signal Processing Architectures*, Vol. 614, 1986, pp. 24–33.
10. Oklobdzija, V. G., and Ercegovac, M. D. An on-line square root algorithm. *IEEE Trans. Comput.* C-31, 1 (Jan. 1982), 70–75.
11. Owens, R. M. Compound algorithms for digit online arithmetic. *Proc. 5th Symposium on Computer Arithmetic*, May 1981, pp. 64–71.
12. Trivedi, K. S., and Ercegovac, M. D. On-line algorithms for division and multiplication. *IEEE Trans. Comput.* C-26, 7 (July 1977), 667–680.

A PROPOSAL FOR THE SYSTEMATIC DESIGN OF ARRAYS FOR MATRIX COMPUTATIONS

Jaime H. Moreno

May 1987
CSD-870019

A Proposal for the Systematic Design of Arrays for Matrix Computations

Jaime H. Moreno
Computer Science Department
University of California, Los Angeles

Report No. CSD-870019*
May 1987

*This research has been supported in part by the Office of Naval Research, Contract N00014-83-K-0493 "Specifications and Design Methodologies for High-Speed Fault-Tolerant Algorithms and Structures for VLSI"

Abstract

We propose to develop a general and systematic methodology for the design of matrix solvers, based on the dependence graph of the algorithms. A fully-parallel graph is transformed to incorporate issues such as data broadcasting and synchronization, interconnection structure, I/O bandwidth, number and utilization of PEs, throughput, delay, and the capability to solve problems larger than the size of the array. The objective is to devise a methodology which handles and relates features of the algorithm and the implementation, in a unified manner. This methodology assists a designer in selecting transformations to an algorithm from a set of feasible ones, and in evaluating the resulting implementations.

This research is motivated by the lack of an adequate design methodology for matrix computations. Standard structures (systolic arrays) have been used for these implementations, but they might be non-optimal for a particular algorithm. Reported systems have used ad-hoc design approaches. Some design methodologies have been proposed, but they do not address many important issues.

A preliminary version of the proposed methodology has been applied to algorithms for matrix multiplication and LU-decomposition. The approach produces structures which correspond to proposed systolic arrays for these computations, as well as structures which exhibit better efficiency than those arrays. The results show that different transformations on a graph may lead to entirely different computing structures. The selection of an adequate transformation is thus directed by the specific restrictions and performance objectives imposed on the implementation. The designer can identify and manipulate the parameters that are more relevant to a given application.

1 Introduction

Matrix computations are the basis for many applications in science and engineering. Examples exist in image and signal processing, pattern recognition, control systems, among others. The evolution in VLSI technology is making possible the cost–effective implementation of many matrix algorithms as a collection of regularly connected processing elements (PEs).

An important problem in the design of arrays of PEs for a given algorithm is the methodology used to derive the structure and interconnection of those arrays. Standard structures (systolic arrays [1]) have been used for these implementations, but they might be non–optimal for a particular algorithm. Ad–hoc design approaches have been applied in the systems that have been reported. Some transformational methodologies have been proposed [2], which either restrict the form of the algorithm (i.e., a recurrence equation), or are unable to incorporate certain implementation restrictions such as number of I/O pads, limited data broadcasting, or lower bound on efficiency.

We have previously devised an algorithmic model and a methodology to evaluate the effectiveness of replication, pipelining and local parallelism in the implementation of multiple–instance algorithms [3]. Such method was used to analyze the Singular Value Decomposition computation, resulting in implementations with significant improvement in efficiency with respect to the linear systolic arrays proposed for it [4].

In this research, we propose to develop a general and systematic design methodology for matrix algorithms, with the capability to handle and relate features of the algorithm and the implementation in a unified manner. This methodology should provide mechanisms to deal with issues such as data broadcasting, data synchronization, interconnection structure, I/O bandwidth, number of PEs, throughput, delay, and utilization of PEs.

The existence of such methodology would allow the designer to identify and manipulate the parameters that are more relevant to a given application. For example, if I/O bandwidth is critical in a certain implementation, the methodology should make it possible to take such requirement into account. In the same way, the methodology should not be restricted to use a particular architecture or interconnection structure.

We propose a methodology based on the dependence graph of algorithms. Starting from a fully–parallel graph, in which nodes represent the operations and edges correspond to data communications, we apply transformations to the graph to incorporate the issues indicated above. The specific transformations depend on the particular parameters of interest. The proposed methodology addresses multi–instance and single–instance computations. To achieve these objectives, some transformations exploit pipelining of data to enhance concurrency and reduce communication requirements, while other transformations are oriented to reduce the computation time.

We suggest to use a fully–parallel dependence graph as the description tool because such notation exhibits the intrinsic features of an algorithm. These graphs are characterized by having all inputs and outputs available in parallel, and no loops (i.e., loops are unfolded). From such dependence graph it is possible to derive an implementation by assigning each node of the graph to a different processing element (PE), and by adding delay registers to synchronize the arrival of data to the PEs. The resulting structure exhibits minimum delay (determined by the longest path in the graph) and optimal throughput (for multi–instance computations), but may require complex or expensive interconnection structure and I/O bandwidth, and large number of units. The suggested

methodology deals with these problems, while still attempting to preserve the features inherent in the dependence graph.

We have applied a preliminary version of this methodology to the algorithms for matrix multiplication and LU–decomposition. We use transformations which incorporate a subset of the design issues listed above. Although some of these transformations seem to be of general application, others seem appropriate only for specific cases. The proposed research is oriented towards identifying and providing a formal definition of a general set of transformations.

In the next section, we define the specific problem that this proposed research addresses. In section 3, we review previously proposed methodologies for the design of arrays, pointing out their major benefits and disadvantages. Section 4 describes the basic transformations in a preliminary version of our systematic methodology, and illustrate the use of these transformations by applying them to the algorithms for matrix multiplication and LU–decomposition.

2 Scope of the Proposed Research

The problem of devising a systematic design methodology for arrays of processing elements has been studied by several researchers, but it remains unsolved in many aspects [2]. As the problem is quite large and complex, it is necessary to focus on a subset of important issues, and address those issues specifically. We indicate now the scope of the proposed research.

2.1 Areas of Application of the Methodology and Admissible Algorithms

In a review of parallel processing algorithms and architectures for real–time signal processing, Speiser and Whitehouse [5] have shown that the major computational requirements for many important real–time signal processing tasks can be reduced to a common set of basic matrix operations. For example, critical signal processing tasks include adaptive filtering, data compression, beamforming, and cross–ambiguity calculation. For these applications, the basic set of required matrix algorithms includes matrix–vector multiplication, matrix–matrix multiplication and addition, matrix inversion, solution of linear systems, eigensystems solution, matrix decompositions (LU–decomposition, QR–decomposition, singular value decomposition). Since these matrix operations provide a large portion of the burden for real–time signal processing, such burden has limited the adoption or even the comprehensive evaluation of new signal–processing algorithms, permitting them to be applied only to small problems in off-line computations, or to limited data sets [5].

Thus, this research is oriented to fulfill needs in the area of high–speed implementations of matrix computations, for fields such as signal and image processing, pattern recognition, and control systems.

The objective of this research is to devise a systematic design methodology for arrays of PEs for *matrix computations*. We have selected matrix algorithms for the following reasons:

i) Matrix algorithms are compute–bound, requiring concurrent implementations to achieve high computation rates.

ii) Matrix algorithms can be decomposed into regular subcomputations, thus they are suitable for implementation as a *collection of regularly connected PEs.*

iii) Matrix algorithms scale regularly, so that implementations can be devised for small matrices and the results extended to larger ones.

iv) Real-time implementation of matrix algorithms are highly desirable, as discussed in section 2.1.

2.2 Design Methodology Based on Graph Transformations

In the proposed methodology, the original representation of the algorithm (a graph) is *transformed* into a form suitable for implementation (the properties of the graph are discussed later). Thus, our methodology follows a transformational approach, the same as other previously proposed schemes. Transformations are guided by implementation restrictions which render the original graph not feasible as an implementation. The objective is to provide a unified treatment of algorithm features and implementation restrictions.

Our method deals explicitly with the restrictions existing for a given implementation. Those restrictions include I/O bandwidth, utilization of PEs, limited data broadcasting, interconnection and communication structure, number of operation units. The designer must select from such restrictions those which are relevant for a particular implementation, resulting in different alternatives depending on the restrictions selected and the order in which those restrictions are taken into account. This is in contrast to other proposals, where emphasis is placed mostly on interconnection regularity and strictly nearest-neighbor communications. The remaining implementation issues are largely ignored during the application of those methodologies, although for a given algorithm such issues might be as important as the ones considered.

2.3 Number of Instances of the Computation

An important parameter that influences a suitable implementation is the *number of times* that an algorithm is executed on independent data. Pipelined implementations are effective only in cases where this number is large with respect to the number of stages in the system. On the other hand, if the algorithm is computed just once, only parallelism is effective to reduce the computation time. Thus, we must distinguish between implementations for multi-instance and for single-instance algorithms, and deal with them differently.

The proposed methodology addresses multi-instance and single-instance computations. Some of the transformations exploit pipelining of input data and of intermediate results, to enhance concurrency and reduce communication requirements. In these cases, the objective is to increase throughput. Thus, the target of those transformations are multi-instance algorithms, or computations which can be decomposed into multiple instances of basic algorithms. Other transformations are oriented to reduce the computation time of the algorithm, and are appropriate for a single evaluation of the computation. The selection of suitable transformations depends on this degree of multiplicity.

2.4 Degree of Automation in the Design Process

For a transformational methodology, an important issue is the degree of automation desired in the process. Many methodologies proposed in the literature state as their goal to completely automate the design process. However, selecting a good transformation at any step in the process is not an obvious task, unless the number of choices is small and they can be evaluated exhaustively. Straightforward transformations may fail, so that the synthesis procedure usually requires creativity from the designer.

The proposed design methodology is oriented towards assisting the human designer, providing him/her with a flexible tool able to incorporate the design requirements. The design process is an iterative, perhaps interactive, procedure in which the designer selects a transformation from a set of feasible ones, applies it, and evaluates the result according to some performance and cost measures defined for the implementation. In this way, the design becomes a search in the space of solutions available through the transformations, for the alternative which offers the best cost-performance trade-offs. This search is assisted by the methodology.

2.5 Model of Computation for the Implementation

Our methodology uses a synchronous model of computation for the implementation, that is, all cells in the array operate synchronously, and all data transfers are performed simultaneously. There are no pre-defined restrictions regarding interconnection and communication structure, or I/O data bandwidth. Thus, the methodology is not restricted to a particular architecture.

2.6 Criteria for Optimality and Evaluation of Implementations

Criteria for optimality and evaluation of implementations are controversial, since the definition of optimality may not be the same for every implementation. The design of VLSI circuits has used A, T, AT and AT^2 as measurements of efficiency (where A is area and T is time) [6],[7]. These measures have been used as "lower bounds to solve problems in the VLSI domain" [6].

If an array of processing elements was to be fabricated as a single VLSI device, then the use of these measures would have the same value as for any other VLSI design. However, it is usually the case that implementations of arrays with current technology do not fit into a single VLSI circuit for most practical examples. Instead, these implementations are arrays of devices, with one or several components per PE [8], [9], [10]. In this context, area-time tradeoffs are no longer as meaningful as in the VLSI domain. Furthermore, area might be a complex function of the number of PEs, number of buffers, interconnection pattern, etc. [11]. Instead, significant parameters may be number of devices (or PEs), throughput, I/O bandwidth, or others. In spite of these facts, the measures mentioned above have been used to evaluate systolic structures [11], [12], where area has been computed as number of PEs. Note, though, that the advent of wafer scale integration (WSI) may bring the former measures back into adequate use.

We suggest to avoid pre-defined criteria for optimality. Our approach to this issue is to provide in the methodology enough flexibility so the designer can choose the measures of interest for the implementation, and evaluate such implementation according to those measures. In this manner,

the methodology exhibits adaptability for different technologies as they evolve in time.

Thus, we propose to devise a methodology in which measures of efficiency and optimality in the design are selected by the designer, from a predefined set, tailoring these parameters to the implementation at hand. This set of measures includes delay (i.e., computation time), throughput, number and utilization of operation units, speed-up, efficiency (i.e., speed-up over number of units). The methodology should provide a mechanism to define such measures, and procedures to evaluate with respect to those measures different designs resulting from the methodology.

2.7 Description of the Algorithms

The description of an algorithm is very important, since the outcome of the methodology depends on it. *Possible description tools* are mathematical expressions, program with loops, program in a parallel high-level language, graphical description (we discuss later the impact of the description of the algorithm on the methodologies proposed in the literature). Our approach uses a graphical description of the algorithm, as indicated below.

Fully-Parallel Graph as Description Tool

We have selected a fully-parallel dependence graph to describe the algorithms. In such graph, nodes represent the operations and edges correspond to data communications. All inputs are assumed available simultaneously, and all outputs are available as soon as they are computed. Loops are unfolded completely, and branches are expressed explicitly.

The graph doesn't include synchronization of the arrival of data to the nodes. Thus, nodes fire when their operands become available, same as data-flow graphs. The major difference between data-flow graphs and fully-parallel graphs is the absence of "tokens" in the latter. These tokens are not needed, since the model of computation we use for implementation is synchronous, and the arrival of data to the nodes is synchronized as the result of graph transformations during the application of the methodology.

We select to use a fully-parallel dependence graph to describe the algorithms, because such notation exhibits the intrinsic features of an algorithm. From the dependence graph it is possible to derive an implementation by assigning each node of the graph to a different processing element (PE), and by adding delay registers to synchronize the arrival of data to the PEs. Figure 1 shows an example of this approach. The resulting structure, which is a *pipelined implementation of the graph*, has the following features:

- minimum delay (determined by the longest path in the graph)
- it provides a new result every cycle (throughput $T = 1[evals/cycle]$)
- optimal utilization of PEs when computing multiple instances of the algorithm

However, the pipelined implementation of the graph may require complex or expensive interconnection structure and/or I/O bandwidth, and large number of units. The proposed methodology deals with these problems, while still attempting to preserve the features inherent in the dependence graph.

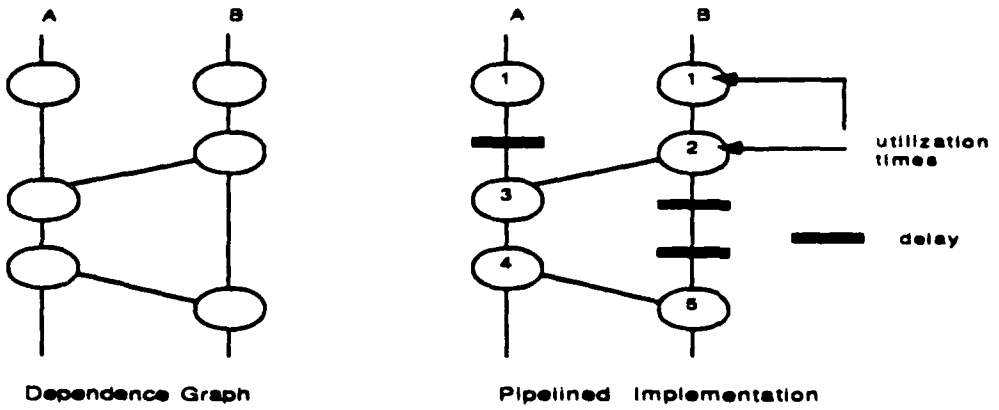

Figure 1: A dependence graph and its pipelined implementation

2.8 Mapping of Problems Larger than Array Size

Computational arrays are usually characterized by the size of the array, which defines the size of the problems that can be solved in such arrays. Many applications often require the processing of large matrices, but it is not feasible to build an array as large as the dimension of such matrices. A similar situation arises when the same array is used to solve problems of different size. In these cases, it becomes necessary to decompose the problem into subproblems, so that the subproblems fit into the target array. This is known as the *partitioning problem.*

We expect to include the partitioning problem in our methodology, since we envision partitioning as an integral part of it. The dependence graph of an algorithm is drawn for the size of the problem, not for the size of the target array. In this way, all data dependences are available to the designer to manipulate them in the most convenient manner.

2.9 Formal Definition of Transformations

Since the proposed methodology is a transformational one, it is necessary to identify a general set of transformations. In addition, it is extremely important that the transformations are proven correct, since the methodology relies on them to achieve its objectives. Thus, it must be proved that for any algorithm the application of a transformation on a dependence graph preserves the function which is expressed by such graph. If such proof exists, the implementations resulting from the application of the methodology will not need to be verified. An implementation is obtained as the result of applying correctness–preserving transformations, and the sequence of transformations itself serves as a constructive proof of correctness of the implementation.

As a consequence, a formal definition and proof of correctness of each transformation is necessary. We expect to achieve this formalism using some symbolic representation of the transformations, so that the same symbolism may be used to prove correctness. Methodologies which use mathematical expressions as the description tool have this property already included. Those that follow other transformational approaches have also had to deal with this issue. Note that we advocate the use of the formalism only to prove the correctness of the transformation, and not as part of the methodology.

In [13], Sheeran uses graphical descriptions to illustrate some transformations formally defined in μFP, an algebraic VLSI design language based on functional programming. We envision an approach in the opposite direction, namely each transformation on the graph has an algebraic counterpart that proves it correct.

We have presented the scope of our proposed research, indicating our approach to face the problem. In the next section, we review previously proposed methodologies for the design of arrays of PEs, and look into how those proposals relate with our approach to the problem.

3 Previously Proposed Methodologies for the Design of Arrays

The development of a systematic methodology for the design of arrays has been actively pursued recently. In [2], Fortes et al. review seventeen different methods for the design of algorithmically specified systolic arrays, and new ones have been suggested since then. Fortes et al. conclude that the most common characteristic of the proposed methods is the use of a *transformational approach*, i.e., "systolic architectures are derived by transforming the original algorithm descriptions that are unsuitable for direct implementation." In other words, the proposed systematic design methods consist of transformations of a high-level specification of a problem into a form better suited for implementation.

Although the proposed approaches can be useful to accomplish certain design tasks, they have limitations. The methods are, in general, unable to incorporate implementation restrictions such as number of I/O pads, lower-bound on the utilization of processing elements, or limited data broadcasting. Furthermore, the success in achieving their goal is not convincing as suggested by Fortes et al., who state that "from a global point of view, it is clearly indicated that the two largest limitations in the state of the art of existing transformational systems are the non-existence of powerful systematic semantic transformations and the inability to systematically achieve optimality in the resulting designs" [2].

In this section, we review previously proposed methodologies for the design of arrays. First, we discuss some classifications of those methodologies, and later look into the impact of the algorithm description in the resulting implementations.

3.1 Classifications of Methodologies

There are several alternatives to classify methodologies for the design of arrays of PEs. Fortes et al. [2] characterize existing transformational approaches as follows:

i) How algorithms are specified, that is, what mechanism or tool is used to present the algorithm to the transformational methodology. This can be a high-level language description, pseudo-code, mathematical expressions, or even a natural language description.

ii) What formal models are used, that is, what representation is used to abstract the relevant features of an algorithm. This model can be based on the functional semantics of an algorithm

(i.e., algebraic expressions), on the structure of an algorithm (i.e., a graph), or a geometric description of the algorithm.

iii) How systolic architectures are specified, that is, what hardware model or level of design is used to describe the array.

iv) What types of transformations are used on and between the representations. That is, whether transformations are systematic or ad-hoc, and whether they are used to go from one type of representation to another, or to the same type of representation but at a different level.

Emphasizing on how and at which level the proposed transformations are applied, Li and Wah [11] classify the methodologies as follows:

1. Transformations performed at algorithm representation level, and direct mapping made from this level to architecture.
2. Transformations performed at algorithm-model level (i.e., at the level corresponding to item (ii) above), with procedures for deriving the model from the algorithm representation and for mapping the model into hardware.
3. Transformations performed on a previously designed architecture to obtain a new architecture.
4. Transformations performed to map a systolic architecture into the function implemented, and to prove the correctness of the design.

From these classifications, we can recognize the importance of the description of an algorithm. Description tools being used are:

- mathematical expressions
- graphical descriptions
- loops and begin-end blocks
- programs in high level language

We look now into how these description mechanisms have been used on the different methodologies. We can anticipate that although some transformations available with those descriptions may be powerful, the resulting schemes are suitable only for the classes of algorithms which have representations as required by the corresponding methodology (i.e., uniform recurrence equations, canonical algebraic expressions, recursion equations, program with loops). Furthermore, it is not always clear whether adequate transformations can always be found with those descriptions, and the target architectures may be restricted as a result of those tools.

3.2 Algorithm Descriptions

We focus our discussion on the impact that the description of the algorithm has on the results obtained with the proposed methodologies. In most cases, the objectives pursued by the transformations are the same, namely to eliminate data broadcasting, and enhance local communications

and regularity. Only the form of the transformations is different, since the representations are different.

Our methodology is not too distinct in this respect, since we also apply transformations to a description of an algorithm (a graph), with similar objectives. Consequently, some of the underlying ideas in the methodologies discussed here are suitable to our approach. The main difference relies in our attempt to incorporate most implementation restrictions, if not all, as part of the design process. This is in contrast to the other schemes, where implementation issues such as I/O bandwidth, for instance, are a result of the application of the methodology, with no control over it.

Methodologies that require the algorithms to be described in a particular manner are limited to those algorithms that have such a description. That is the case, for instance, of those schemes which use recurrent expressions. Since our approach is more general than that, requiring only to draw the dependence graph of the algorithm, it might be the case that for certain specific classes of algorithms other methodologies are more convenient. We have no definite answer regarding this issue yet. Thus, our discussion below about previously proposed methodologies doesn't address this topic.

Mathematical Expressions

The manipulation of mathematical expressions to derive arrays of PEs allows to apply powerful algebraic transformations to an algorithm. These transformations attempt to enhance pipelining and local communications by index transformations, that is adding indices to existing variables, renaming variables, or introducing new variables. The resulting expressions are in some "canonical" form, which corresponds to structured sets of computations written as recurrence relations or nested loops [14].

Kung and Lin, for example, have proposed an algebra for systolic computation [15]. Algebraic transformations are applied to the algebraic representation of an algorithm to obtain an algebraic representation of a systolic design. They use this algebra to derive designs which have what they call the "systolic property," that is, designs where broadcasting has been replaced by distributing common data to different destinations at different times. Such design is represented as a Z-graph, where there is an edge for each variable and a node for each computation. Edges have labels of the form z^{-k}, where k denotes the delay between nodes. The Z-graph representation is readily mappable to hardware.

Guerra and Melhem [14] address the design of systolic systems with non-uniform data flow, based on a subset of the data dependences extracted from the original problem specification. Their approach identifies chains of dependent computations which are converted into recurrence equations, and then maps the new specification into hardware.

Li and Wah [11] formulate the design as an optimization problem, where the search space is polynomial on the problem size. They propose a methodology for the design of optimal pure planar systolic arrays for algorithms that are representable as linear recurrence processes, using completion time T or area-time AT^2 as figures of merit for the design. Their systolic designs are characterized by three parameters, namely velocities of data flows, spatial distribution of data, and periods of computation. They express completion time and hardware complexity in terms of these parameters.

Quinton [16] uses a set of uniform recurrent equations over a convex set D of cartesian coordinates as the description of the algorithm. The proposed methodology first finds a timing function compatible with the dependences of the equations, and then maps the domain D into another finite set of coordinates. However, it is not clear how the mapping is systematically performed. Furthermore, the method is limited to cases where one of the data streams is dependent on other data streams [17].

Weiser and Davis [18] propose a method for treating sets of data as wavefronts entities, and apply transformations to the wavefronts. The problem is presented as operations on sets of data, using a KM function on such data. Thus, wavefronts, KM functions and the transformations need to be identified. The method is best applicable to algorithms that can be described by relatively simple and concise mathematical expressions [2].

The nature of the mathematical expressions used in these methods may lead to inadequate conclusions regarding the features of an algorithm. A significant example of this situation is found in the pioneering work of Kung [19], where he concluded that "LU–decomposition, transitive closure, and matrix multiplication are all defined by recurrences of the "same" type. Thus, it is not coincidental that they can be solved by similar algorithms using hexagonal arrays." A similar statement is found in [20]. However, matrix multiplication and LU–decomposition have quite different dependence structures, so that they are not mapped efficiently into similar arrays [21].

Furthermore, schemes using mathematical expressions are suitable only for the classes of algorithms which have representations as required by the corresponding methodology (i.e., uniform recurrence equations, canonical algebraic representations, recursion equations).

Loops

Program with loops as the starting point of a methodology have been used in [20] and [22]. In [20], Moldovan first "transforms the algorithm with loops into a highly–parallel form suitable for VLSI, and then transform the resulting algorithm into a systolic array." The idea is that the data dependences of the new structure can be selected in the transformation process. Unfortunately, this work does not describe a systematic way to find the transformation [16]. Moreover, it is suited only to the class of algorithms which can be described by programs with loops (or recurrence equations) [2].

Miranker and Winkler's work [22] is similar to Moldovan's approach. Extensions are the rewriting of expressions by using properties of the operators in an ad–hoc manner, and the use of graph embeddings based on the longest path of a computation graph when such graph is too irregular. Theoretically, this approach applies to any algorithm, although systematic design seems possible only for those algorithms described by programs with loops [2].

Moldovan's work brings up an important issue which has also been recognized as such by other researchers: the data dependences are the clues to potential transformations [17]. Moldovan's approach uses this fact to perform linear transformations on those dependences, with the main goal of eliminating data broadcasting and global communications. Our approach also uses the data dependences, although we do not require the algorithm to be described in terms of loops.

Graphical Descriptions

Since data dependences in an algorithm contain relevant information required for an implementation, it seems appropriate to use those dependences as the description of the algorithm, and the starting point of a design methodology. We discuss now some approaches in this direction.

Ramakrishnan et al. [23] proposed a formal model for a linear array of processing elements, and graph representations of programs suitable for execution on such a model. These graphs were defined as homogeneous graphs, which are a more limited class of program graphs than general data–flow graphs. In particular, homogeneous graphs have the same number of edges into and out of every node, excepting those nodes representing sources or sinks of data (i.e., inputs or outputs, respectively). The proposed methodology first partitions the graph into sets of vertices that are mapped into the same processing element. In a second step, a syntactically correct mapping is used to map computation vertices onto processors and time steps, and labels and edges to *communication delays* and *interconnections* [2]. This methodology applies only to homogeneous graphs with connected subgraphs satisfying certain properties. Moreover, the method can only be used to generate linear arrays [2].

Another graphical approach has been pursued by Barnwell and Schwartz [24]. This method starts with an algorithm described as a fully–specified flow graph, that is, a directed graph in which nodes represent operations and edges represent signal paths. Nodes are also used to represent delays explicitly, when those delays are part of the algorithm (e.g., digital filters). This description tool seems to be almost the same as the fully–parallel graph we use in our approach. The major differences are the requirement on the nodes to be fundamental operations performed by the cells in the implementation, and the fact that this approach is oriented to implementations with identical cells. The latter characteristic arises as a result of targeting the methodology to implementations in multiprocessors.

Barnwell and Schwartz use two different models of computation: a synchronous model, which they associate with systolic arrays, and a "skewed single instruction multiple data" model (SSIMD), where the same program is executed in each cell in the array, and that program realizes exactly one time–iteration of the flow–graph. In other words, they exploit the multi–instance property of algorithms to allocate a separate processor to each instance of the computation. We have provided a more formal characterization of this situation when comparing implementations using replication and pipelining for multiple–instance algorithms [3].

Barnwell and Schwartz claim that since systolic arrays are characterized by synchronous data transfers, flow–graphs are constrained to have every output from a cell terminated by a delay node (or pipeline register). Hence, "the generation of systolic solutions for flow–graphs reduces to the distribution of delays nodes throughout the flow–graph." Their methodology consists of systematic manipulation of the flow–graphs into systolic forms, using theorems from graph theory. Although their assertion is true, the methodology is not as general as we claim necessary since it doesn't deal explicitly with other important issues such as input/output bandwidth, limited data broadcasting, or lower bound on efficiency. Furthermore, their transformations are ad–hoc instead of systematic [2].

Jover and Kailath [25] proposed a pseudo–graphical approach. They introduced the concept of *lines of computation LOCs*, which are useful to determine whether a given topology is suitable for systolic computations. *LOCs* are a summary of an architecture in time or space, and some

properties of the architecture can be inferred from such *LOCs*. Jover and Kailath's work includes the definition of *systolic-type arrays*, a generalization of systolic arrays where there may be different cells not only at the boundary of the array, but also inside. This idea deserves to be considered, since implementations may benefit from not being restricted to identical cells. However, *LOCs* are not general since they require the computation of the algorithm to be distributed through the array (i.e., they do not allow a result to be "accumulated" locally in a cell). Moreover, *LOCs* are not really a design methodology.

Approaches Using High-Level Languages

General purpose high-level languages have also been used as the starting point of design methodologies. Lam and Mostow [26] use a high-level algorithm description language, and model the design process as a series of transformations on this description. They rely on the human designer to decide which transformation to apply, instead of aiming towards a fully automated approach. A computer-aided transformational tool is used to assist the designer in the process. The design process first performs software transformations on the description of the algorithm to prepare it for systolic implementation. This initial transformation converts the algorithm into a representation composed of highly repetitive computations, expressed in terms of nested-loops and begin-end blocks. This step includes the annotation of the algorithm description with statements to indicate how subfunctions should be evaluated, such as "in parallel" or "in place" (i.e., sequentially within the same unit). The automated software tool knows how to map such constructs onto hardware. Then, a sequence of hardware allocation, scheduling and optimization phases are applied iteratively. The optimization phase is guided by the user, who selects the transformations to apply [26]. The method can only process algorithms with simple loops and begin-end blocks, simple unnested function calls, scalar and array variables. It cannot deal with other high-level software constructs [2].

The underlying approach and some of the transformations used in Lam and Mostow's work is quite similar to what we propose for our research. The main difference is due to the representation of the algorithm. Lam and Mostow claim that "approaches that abstract a computation in terms of its data dependencies assume that the computation has already been decomposed into operations corresponding to the behavior of individual cells. This assumption is impractical for complex computations where the design process may depend on details of internal cell behavior." While this may be true for some of the methodologies proposed in the literature, we believe that exploiting those dependencies through a methodology which exhibits them clearly provides more elements to face the design task and achieve suitable implementations.

Chen [17] uses a general purpose parallel programming language as the description tool. An algorithm specified in such language is improved by transformations that remove broadcasting and limit the number of fan-ins and fan-outs. Another phase of transformations incorporates pipelining and attempts to fully utilize the hardware resources. These transformations are algebraic manipulations on the expressions in the parallel programming language, so that the language must be amenable to algebraic modifications. Thus, the approach is highly mathematical and therefore subject to the same limitations as described before for schemes based on mathematical expressions. Furthermore, it incorporates only the implementation issues indicated above.

Chapman et al. [27] use the OCCAM programming language for algorithm description, simulation and eventually implementation. Although the proposed approach yields programs which

could be used on an array of Transputers (the INMOS processor), the objective of this work is the utilization of the OCCAM algebra to aid in the design. Chapman et al. claim that an OCCAM program can be interpreted as an algebraic description of a regular array architecture that implements a given algorithm. The OCCAM program can be transformed and, as long as the algebraic rules are adhered to, the designer can assume that the program will implement the same algorithm. However, there is no systematic way to perform those modifications, neither a mechanism to enforce only valid transformations.

3.3 The Partitioning Problem

The methodologies proposed in the literature have not explicitly considered mapping of problems larger than the size of the array. This problem has been studied extensively in other contexts, and ad-hoc solutions have been suggested in a manner similar to the design of arrays [28], [29], [30], [31]. These schemes basically assume the existence of an array, and partition the problem to map it into such an array. In the process, they may resort to transformations of the original problem so that it fits in the array (e.g., transforming a full matrix into a band matrix). These approaches require extra computational work to first decompose the problem, and then to combine the results from the component parts since they may overlap. Alternatively, such schemes may require different types of arrays to implement the sub-algorithms that compose the original problem. Our approach to the partitioning problem is different, since we intent to incorporate it as part of the design methodology.

3.4 *Criteria of Optimality*

Most proposed methodologies do not deal explicitly with optimality. They address some implementation issues, in particular nearest-neighbor communications and regular structure, without further evaluation of the results. In general, the methods are unable to systematically achieve optimality in the resulting design [2].

However, a few papers have addressed the topic of optimality in the design [11], [12]. For example, Li and Wah [11] proposed a methodology which measures the merit of a design in terms of the computation time (T), or the product of the VLSI area and the computation time (AT), or the product of the VLSI area and the square of the computation time (AT^2). The effectiveness of these measures has been discussed in section 2.6.

Ramakrishnan and Varman [12] present a family of linear-array matrix multiplication algorithms on a pipelined linear array. These algorithms exhibit a tradeoff between the number of processing cells and the local storage in a cell. However, the total time and storage requirements remain invariant in this tradeoff. This work provides different schemes to multiply matrices in linear arrays, all with the same area (defined as the product of the number of cells and the storage per cell), and the same computation time. However, it doesn't address the issue of optimality for other implementations of matrix multiplication, neither of other algorithms.

3.5 Basic Limitations in Previously Proposed Methodologies

From this review of previously proposed methods for the design of arrays of processing elements, we can conclude that the goal of devising a systematic methodology for this type of structures is far from being achieved. Some limitations of proposed methodologies are easily identified. Among them, we can mention the following ones:

i) Most of the proposed methodologies are oriented towards the design of standard structures (i.e., systolic arrays). However, such structures might be non-optimal for a particular algorithm.

ii) Proposed methodologies do not take into account certain implementation restrictions such as I/O bandwidth, lower bound on utilization of PEs, limited data broadcasting capabilities. They deal mainly with interconnection structure of the resulting arrays, and removal of data broadcasting.

iii) Proposed methodologies are too restrictive regarding the form and the implementation of the algorithms. That is:

- they are suitable only for algorithms which can be expressed in a given form
- they have strict implementation requirements regarding classes of PEs, fan-in/fan-out characteristics, data broadcasting

iv) Description tools used may not convey all the information about an algorithm, leading to inadequate conclusions.

v) Proposed methodologies do not have well defined criterias for optimality in the design.

vi) There is no systematic evaluation of implementation parameters.

In the next section, we present the basic features of a preliminary version of our methodology, which attempts to solve some of the current problems in this field.

4 A Graph-Oriented Design Methodology for Arrays of PEs

We present now the basic features of a preliminary version of a systematic design methodology for arrays of PEs. This is a transformational methodology, which uses a fully-parallel dependence graph as the description of the algorithm. In such graph, nodes represent the operations and edges correspond to data communications. Transformations are applied on the graph to incorporate implementation restrictions. The specific transformations depend on the particular parameters of interest at a given time. The graph used in this approach is called fully-parallel, because all inputs are assumed available simultaneously, and all outputs are available as soon as they are computed. The graph doesn't include synchronization of the arrival of data to the nodes, since such synchronization is accomplished as a result of the methodology.

We have applied this preliminary version of the methodology to the algorithms for matrix multiplication and LU-decomposition. We have used transformations which incorporate a subset of the issues arising in a design. Although some of these transformations seem to be of general

application, others seem appropriate only for specific cases. The proposed research is oriented towards identifying and providing a formal definition of a general set of transformations.

We first describe some transformations to dependence graphs, under certain constrains of interest. Later, we will apply such transformations to specific algorithms.

4.1 Restricting I/O bandwidth

Let's assume that the I/O bandwidth of an implementation is limited, so that it is not feasible to provide the bandwidth needed by the fully-parallel dependence graph. In such case, the entire parallelism available in the algorithm can't be exploited, due to lack of data. Under these circumstances, the implementation of the graph is data-bound.

To take the I/O restriction into account, we modify the dependence graph by *introducing additional nodes to represent the input/output pads.* Since these pads are shared by different edges of the graph, we need to modify the graph so that no contention resulting from the use of the shared resource arises. To achieve this for output pads, we add delay nodes to some of the edges of the graph. For input pads, the data is delayed because it is brought into the computing structure serially through the pads. In this way, we modify the fully parallel-graph by introducing time-multiplexing of the shared resource (i.e., the pads). The application of this transformation has the following characteristics:

- The I/O restriction is applied either to input or output pads first.

  As a consequence of restricting the input pads, the requirements for output pads may decrease since all results may not be available at once. A similar situation is true for input pads if output pads are restricted.

- The assignment of edges of the graph to input/output pads is performed using the length of the paths in the graph as the selection criteria. For simplicity, we describe the methodology for output pads only. This is accomplished as follows:

  - edges belonging to shorter paths are assigned to the output pads first, expecting that longer paths can use those pads at a later time.
  - when paths have identical length, we use the knowledge about the algorithm to carry out the assignment.

Figure 2 shows this class of transformation for the output portion of a given algorithm. In this example, k groups of n outputs each have been multiplexed into a single set of n outputs. Delay nodes have been added to the edges of the graph to avoid contention for the output pads, as indicated above.

Restricting output pads implies that more than one cycle is needed to read results out of the processing array. Similarly, the restriction in input pads forces to use more than one cycle to load the data into the processing structure. Thus, throughput $T = 1[eval/cycle]$ is no longer possible. Therefore, a pipelined implementation of the modified graph will not result in optimal utilization of operation units. Additional transformations to increase such utilization will be described later.

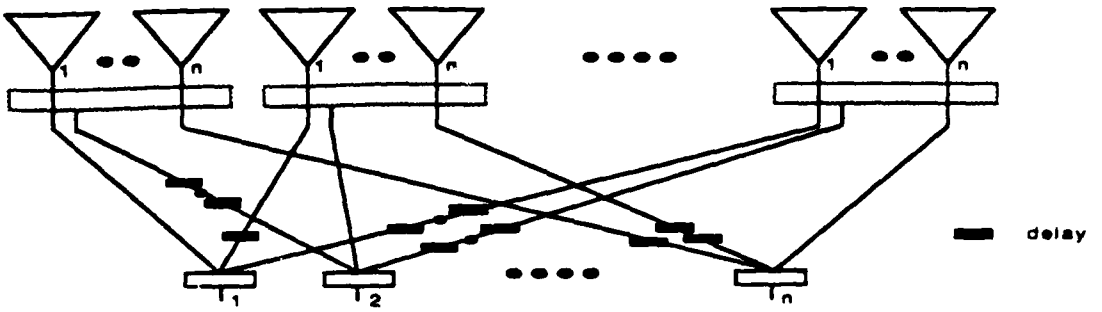

Figure 2: Restricting I/O bandwidth: output pads and delays added to matrix multiplication

4.2 Removing Data Broadcasting

We describe now a transformation to eliminate data broadcasting by replacing it with data pipelining. We apply the following methodology:

- For each broadcasted data element, identify all paths in the graph from the broadcast point to the outputs. Compute the length of each of such paths to the corresponding outputs. Add a new node to the dependence graph, representing a delay element, with its input connected to the broadcasted data. Connect all paths of broadcasted data, excepting the longest one, to the new node. Repeat this process if broadcasted signals remain present in the graph.

  Figure 3a shows this transformation. In this case, two iterations adding delay nodes were required because the topmost broadcasted signal reaches three nodes.

- After all broadcasting has been removed, synchronize the arrival of data to the nodes by adding delay nodes in the shorter paths. This can be achieved by adding the delay nodes using breadth-first search starting from the nodes associated with the data inputs. Figure 3b shows this new transformation.

- Combine computation nodes and delay nodes wherever possible. Replace delay nodes at the input by delaying the input of data itself. Figure 3c depicts this modification.

If a broadcasted data element has equally long paths through the graph, placing the delay nodes would require exhaustive search to identify which paths should not be delayed. Such selection has to be evaluated in terms of the design parameters. To reduce this problem, it is possible to perform the selection taking advantage of knowledge about the algorithm. Such a situation is illustrated in section 4.5, where this kind of transformation is applied to specific algorithms.

4.3 Utilization of PEs

As pointed out in section 4.1, a graph which has been modified by a transformation such as restricting I/O bandwidth might not have throughput $T = 1[eval/cycle]$. In such a case, a pipelined implementation of the dependence graph yields suboptimal utilization of units. It is possible to improve that utilization by *assigning several nodes of the graph to each PE*. To do so, we modify

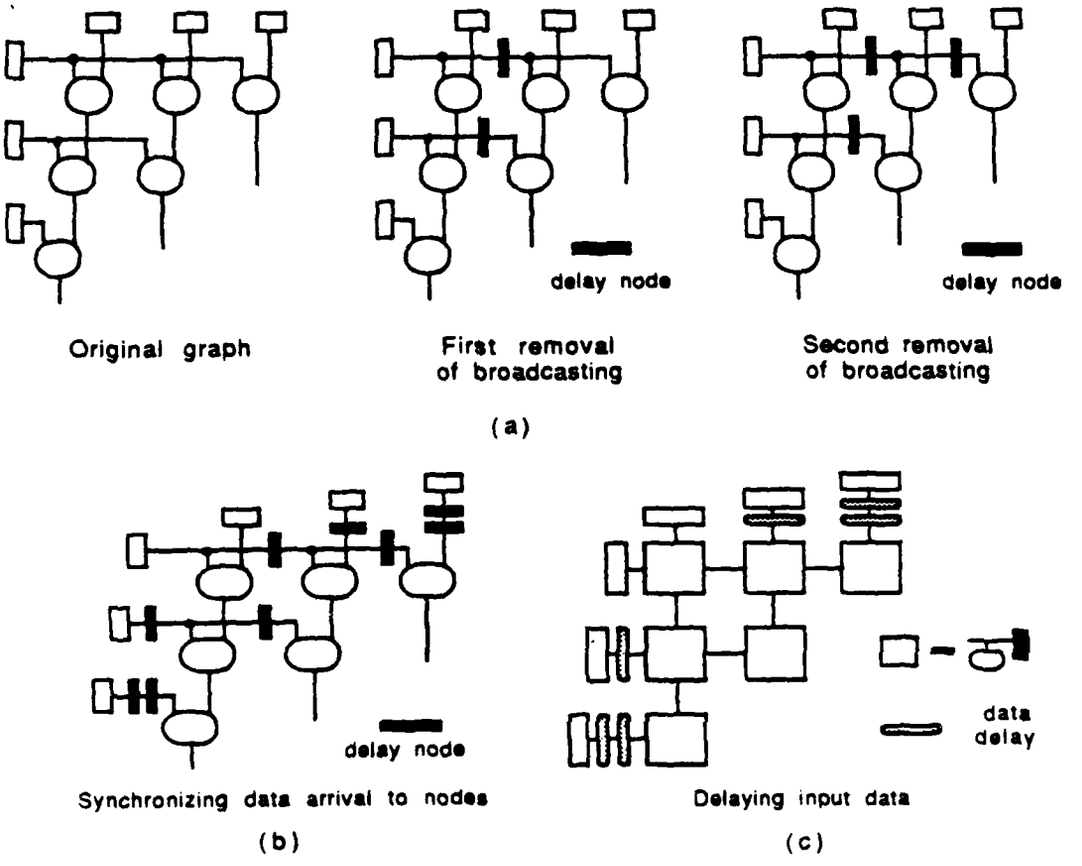

Figure 3: Removing data broadcasting

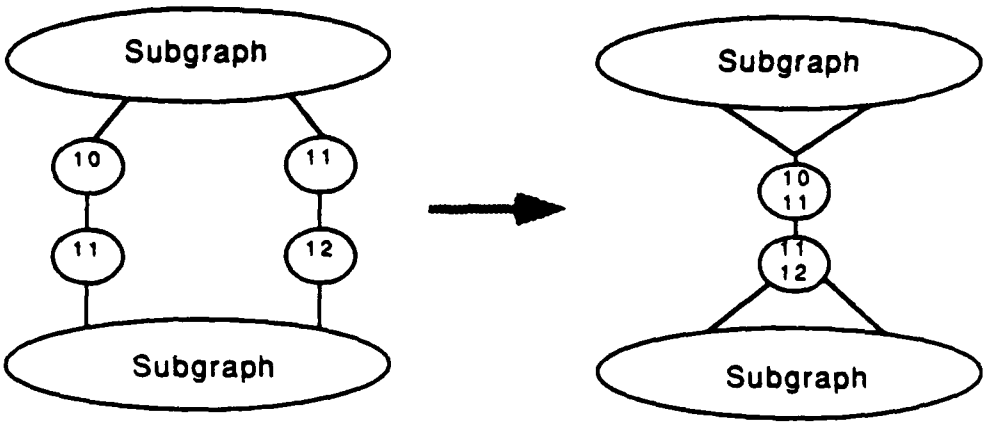

Figure 4: Utilization of PEs: collapsing identical paths

the graph by collapsing groups of nodes into a single node, and assigning the resulting node to a PE.

Our methodology carries out this transformation as follows:

- Annotate each node with the time at which it is used. This information is obtained by traversing the graph from input to output. We assume the same computation time for all nodes.
- Collapse identical paths in the graph which have different utilization time.
- Serialize onto the resulting input path the data used for the different paths which have been collapsed.
- Assign each of the collapsed nodes to a different PE.

Figure 4 shows an example of this approach. The utilization time annotated in the nodes suggest that these two paths can be collapsed as depicted.

To extend the capabilities of this transformation, we may move delays existing in the dependence graph up or downstream. In this manner, we may manipulate the nodes of the graph so that identical paths, suitable for collapsing, have different utilization times. Examples of this situation are shown in section 4.5.

4.4 Removing Input Data Bottleneck

There is another approach towards solving the problems created by a restricted I/O bandwidth. If input pads do not provide enough capability to transfer all needed data at once, the data input process becomes a bottleneck in the implementation. An identical situation is true for output data and output pads. In such cases, it is possible to separate the computation from the transfer of data into or out of the processing structure so that one instance of the algorithm may be under computation, while data belonging to another one is being transferred to/from the system.

Figure 5 depicts this situation. In this figure, three data elements are used three times to

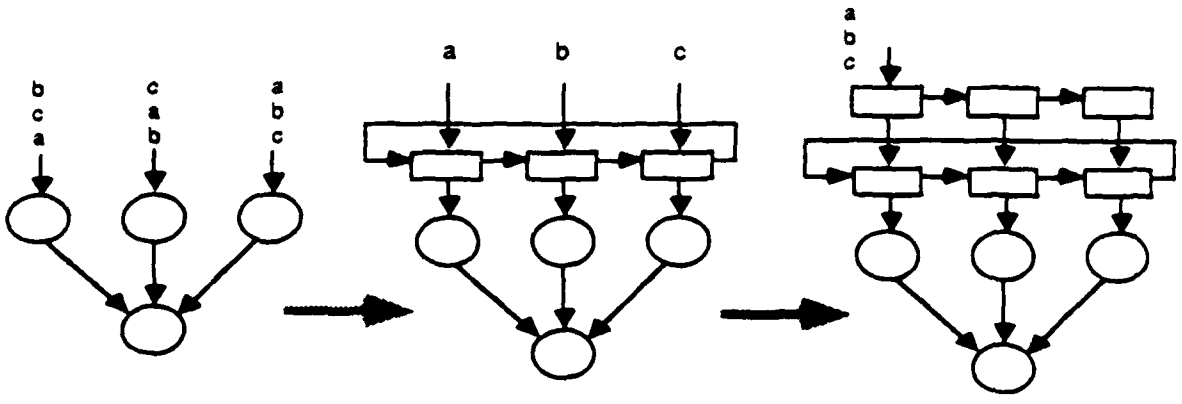

Figure 5: Removing input data bottleneck

perform independent computations in three different units. For the fully-parallel implementation three different input pads are needed, all of them carrying the same data but in a different order. Consequently, the data can be input once, and circulated through a ring structure. If a single input port exists, it can be used to input the data serially through a shift-register structure. After the three elements have been loaded into the shift-register, they are transferred simultaneously to the ring. While these elements are used for computation, a new set of data values is brought into the structure.

The approach requires additional memory elements to store the data, but allows to achieve the degree of parallelism permitted by the dependence graph under a restricted I/O bandwidth situation. Those memory elements can be organized in such a way as to reduce the interconnection requirements of the computation, as shown by the shift-register in Figure 5.

4.5 Application of the Transformations

In this section, we illustrate the application of the proposed methodology to matrix multiplication and LU-decomposition. We show that the application of different transformations to the dependence graph of an algorithm may lead to entirely different architectures. We present the dependence graph of these widely used matrix computations, and use them as target for the application of the methodology outlined here.

4.5.1 Matrix Multiplication Algorithm

The algorithm for multiplication of square matrices is described by the dependence graph in Figure 6. It basically consists of a collection of n^2 inner-product trees. From the graph, we can infer the following properties of the algorithm:

- each input data element is used in n inner-product trees. This indicates that broadcasting of input data is required.
- inner-product trees are independent among themselves: they depend only on the input data.

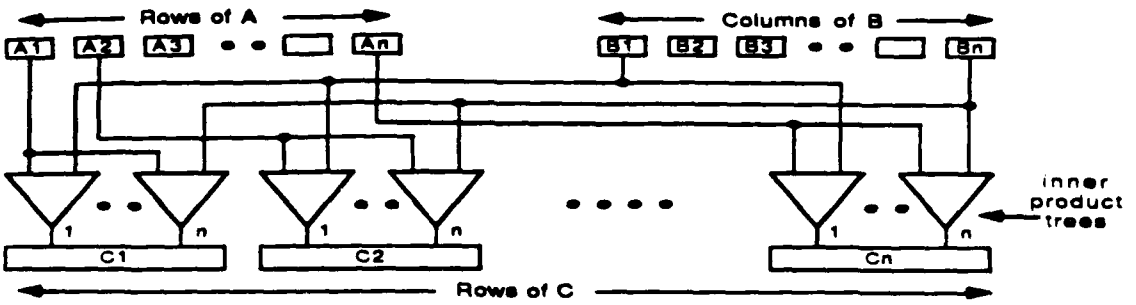

Figure 6: Matrix multiplication dependence graph

- a pipelined implementation of the graph has delay $O(logn)$ and throughput $T = 1[eval/cycle]$. Such implementation requires an I/O bandwidth $O(n^2)$, and broadcasting of the input data.

Several schemes have been proposed for matrix multiplication. Among the most popular implementations, we can list the hexagonally connected mesh proposed by H.T. Kung [19], and the quadratic array suggested by Speiser and Whitehouse [5]. Neither of these schemes has resulted from a systematic evaluation of the various implementation parameters.

4.5.2 I/O and PE Utilization as Main Restrictions in Matrix Multiplication

We assume that the I/O bandwidth is restricted by the implementation to $O(n)$. For simplicity, we consider this restriction as $3n$: one row or column from each of the input matrices, and one row of the result. In such case, it is not possible to exploit the entire parallelism available in the matrix multiplication algorithm due to lack of data. The minimum computation time is now $O(n + (logn))$, since the entire matrix has to be loaded in n cycles. The throughput is now $O(n)$. The implementation of this graph is data-bound.

To take the I/O restriction into account, we modify the dependence graph by restricting output pads first as indicated in section 4.1. We introduce additional nodes to represent the output pads and delays to avoid conflicts, as shown in Figure 7 for a $n = 3$ square matrix. Since all paths in the graph are identical, we use our knowledge about the algorithm and place the delays in main-diagonal-wise fashion (column-wise or row-wise approaches are also possible).

A pipelined implementation of the modified graph has low utilization of units because, in spite of the delays added, all nodes in the graph are evaluated simultaneously but only once every n cycles. To improve the utilization, we move the delays upstream in the dependence graph, as shown in Figure 8. The utilization times annotated in this graph indicate that now nodes are computed at different times. We collapse identical paths with different utilization times and serialize their input data, as proposed in section 4.3. The resulting graph, shown in Figure 9, illustrates the need to replicate data since the same data elements are needed either in different places and/or at different times.

Data duplication may be eliminated in this case in a simple manner, because the interconnection pattern of the replicated data elements is simple. The elements of matrix B are replicated in time so that a single copy of such data is needed, which is accessed repeatedly. The elements of matrix

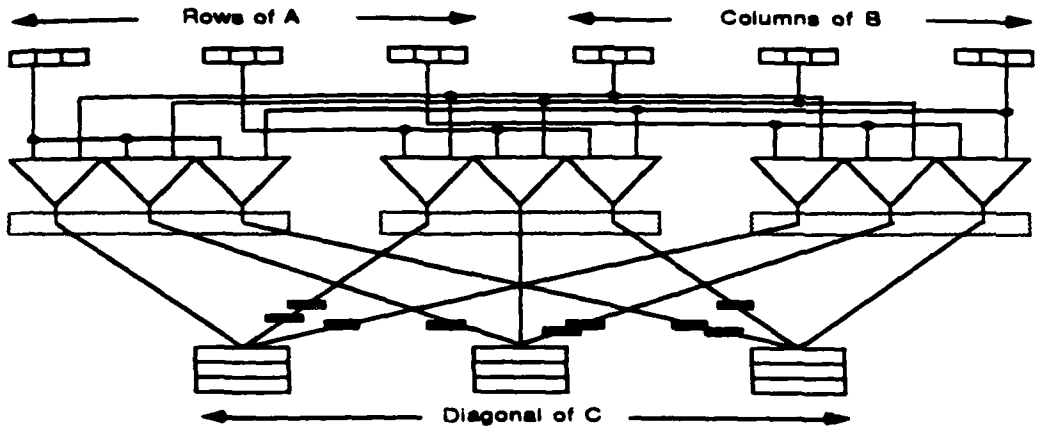

Figure 7: Modified matrix multiplication algorithm for $n = 3$: output pads added

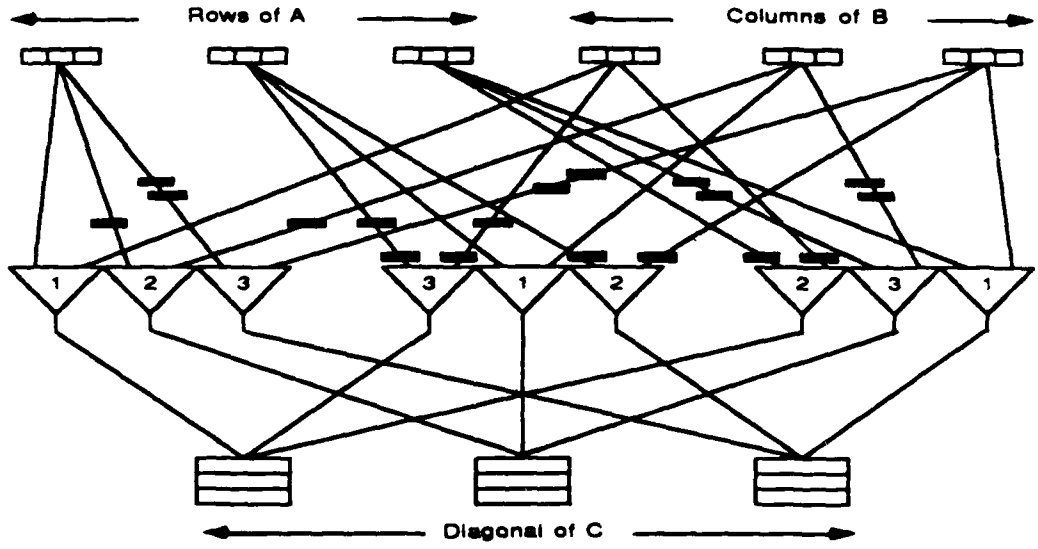

Figure 8: Modified matrix multiplication algorithm: delays moved upstream

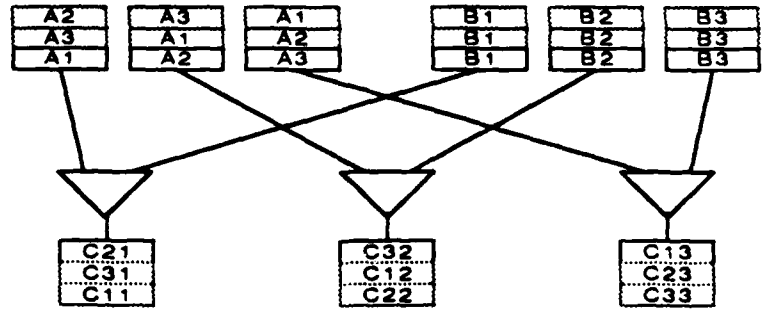

Figure 9: Modified matrix multiplication algorithm: nodes collapsed

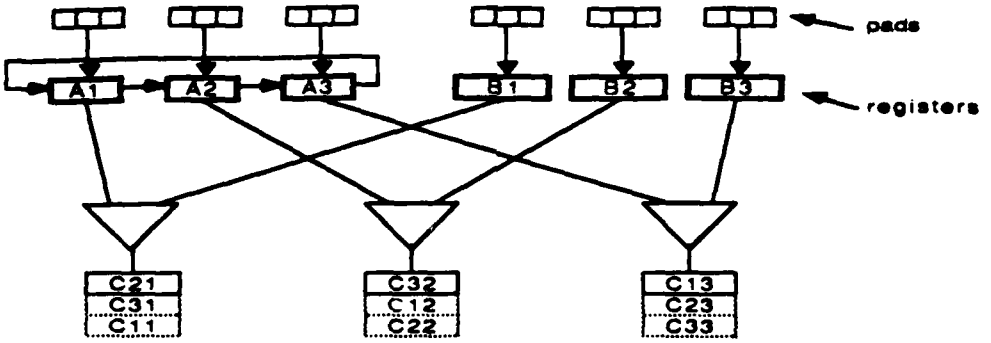

Figure 10: Modified matrix multiplication algorithm: replicated data reduced

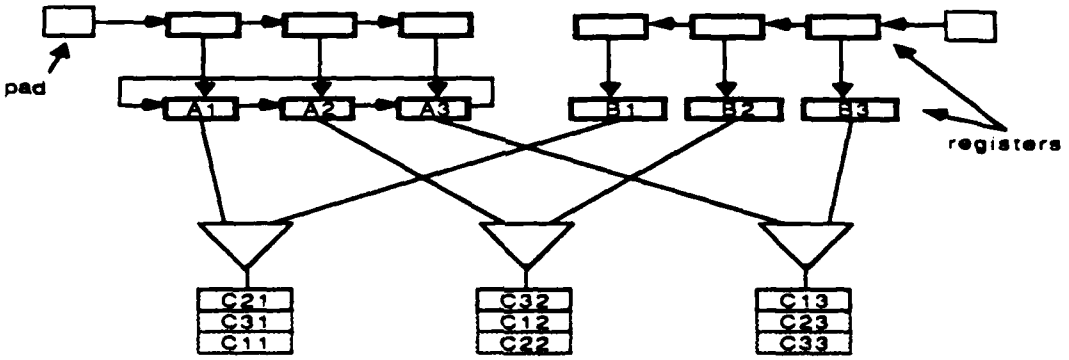

Figure 11: Modified matrix multiplication algorithm: input data bottleneck

A, on the other hand, exhibit a circular–shifting behavior. Storing this data in registers with the same interconnection characteristics allows to keep only one copy of the data. Figure 10 shows the structure resulting after these issues have been considered.

The dependence graph in Figure 10 has input bandwidth $O(n^2)$, but input pads are used only once every n cycles. Reducing the input pads to n produces an input data bottleneck situation, which can be eliminated as suggested in section 4.4; that is, by isolating the input data process from the computation. The modified graph is shown in Figure 11.

A pipelined implementation of the resulting graph has input data bandwidth requirements within the constrains stated, and maximum utilization of the PEs. The operation of the system is as follows: a pair of matrices is loaded in n cycles. After such matrices are stored in the input registers, they are transferred simultaneously to the inner–product trees registers where they are used for the next n cycles. In the meantime, another pair of matrices is being loaded in the input registers. We call this a Multiple Tree architecture for matrix multiplication.

4.5.3 Regularity as Main Restriction in Matrix Multiplication

In this example, we are concerned with regularity in the design, nearest–neighbor communications, no data broadcasting, and identical computation units.

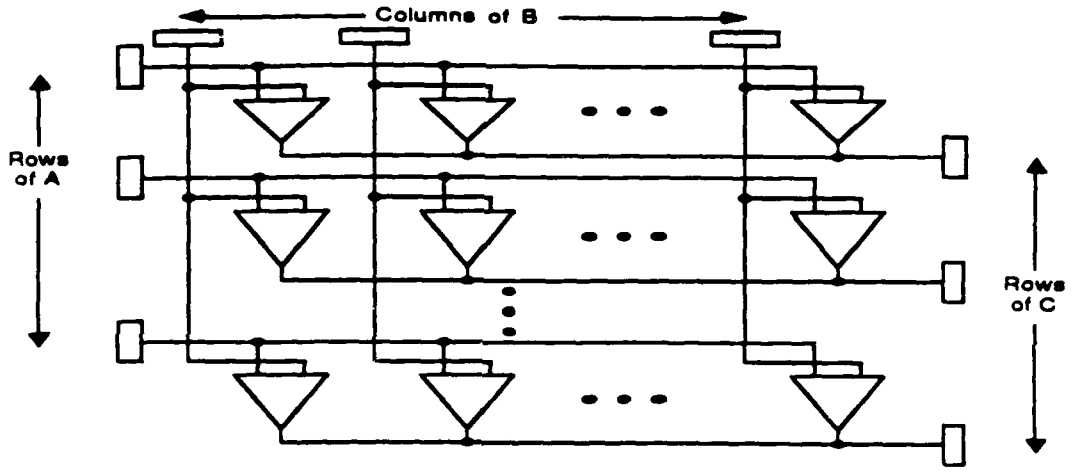

Figure 12: Two-dimensional matrix multiplication graph

Since data dependences in the matrix multiplication algorithm are defined by the two input matrices, we first draw the algorithm as shown in Figure 12. This is exactly the same as Figure 6, but drawn with a two-dimensional input data flow.

In this case, we first attempt to eliminate data broadcasting by replacing it with data pipelining, as suggested in section 4.2. Since broadcasted data paths are of the same length, we take advantage of the knowledge about the algorithm and place the delay nodes in row-wise and column-wise ordering. Figure 13 shows the result of applying the method to the graph in Figure 12. An implementation of this graph has input bandwidth $O(n^2)$, computation time $(2n-1)$, optimal efficiency, and throughput $O(1)$.

Let's assume now that the input bandwidth of the implementation is restricted to $2n$. Applying the methodology described in section 4.1, we introduce nodes for the input pads and serialize the data input process. As a consequence, we have created an input data bottleneck, and nodes have different utilization time. We choose to collapse identical paths in the inner-product trees, as shown in Figure 14. (Alternatively, we could attempt to isolate the data input process from the computation, as suggested in section 4.4.) Thus, the parallel-input inner-product trees are transformed into serial-input trees. The resulting graph is shown in Figure 15; it has bandwidth $3n$, computation time $(3n-2)$, optimal efficiency, and throughput n. It corresponds to Speiser and Whitehouse's systolic array for matrix multiplication [5].

Thus, we have shown that a systematic transformation of the matrix multiplication dependence graph allows to obtain implementations with different architectures and performance characteristics.

4.5.4 LU-Decomposition Algorithm

The LU-decomposition computation is described by the dependence graph shown in Figure 16. This graph depicts a varying degree of parallelism at different steps in the computation, and broadcasting of intermediate results. Input data, on the other hand, is used in only one specific place.

From the graph, we can infer that this algorithm has a dependence structure quite different

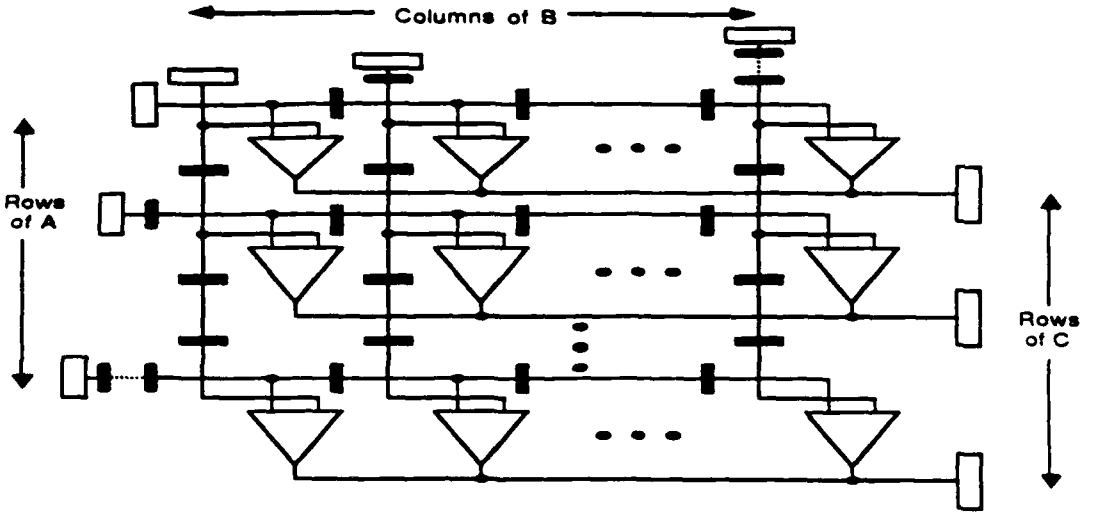

Figure 13: Two–dimensional matrix multiplication graph:delays added

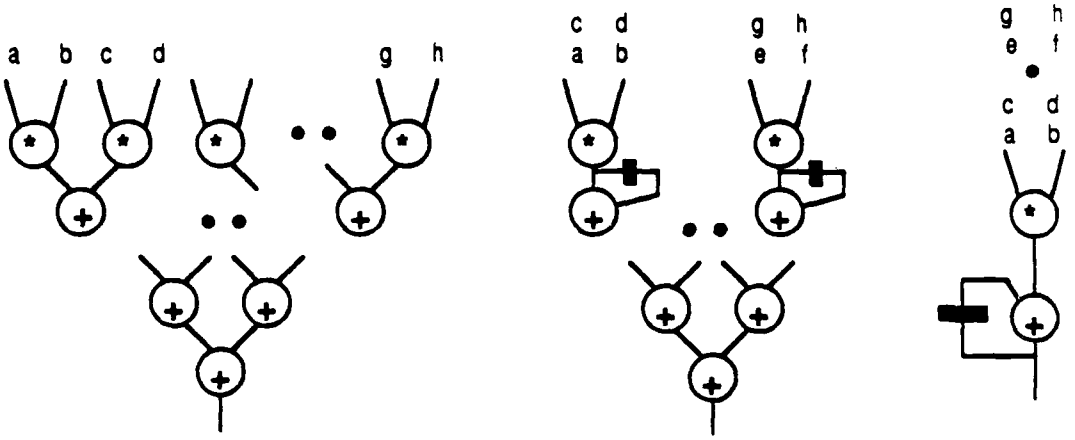

Figure 14: Transformation of inner-product trees

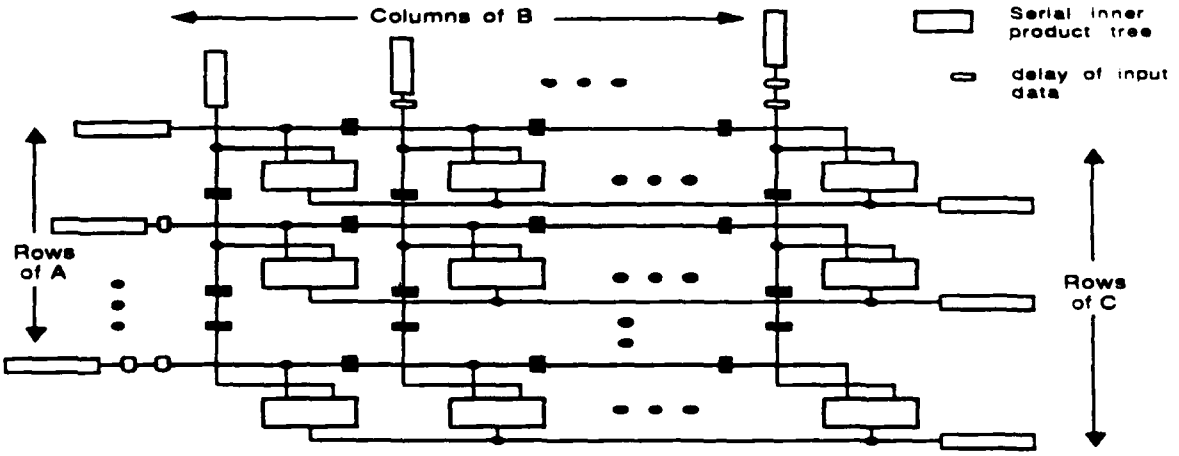

Figure 15: Two–dimensional matrix multiplication graph: serial input trees

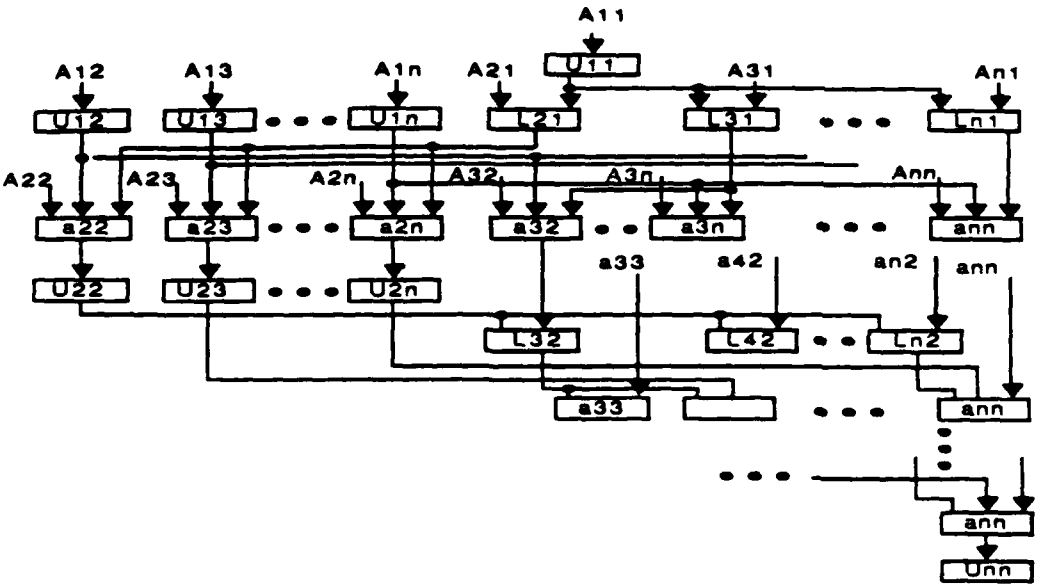

Figure 16: LU–decomposition graph

from matrix multiplication. However, when expressed as recurrence equations, both matrix multiplication and LU–decomposition have similar expressions. This fact has led several researchers to suggest similar arrays for both computations, highlighting their similarity [19], [20]. However, due to the differences, those proposed arrays have low utilization of the PEs.

4.5.5 Data Broadcasting and Utilization as Main Restrictions in LU–Decomposition

In this example, we are interested in the elimination of data broadcasting and in maximizing the utilization of the PEs. We first solve the data broadcasting problem, applying the methodology described in section 4.2. The graph resulting from the application of this transformation to a 5 by 5 matrix is shown in Figure 17, which also indicates the utilization time of the nodes. This graph is characterized by sets of sequential computations (depicted as diagonals of nodes in the figure), which are interdependent. An attractive result of the transformation applied is that data arrives to the nodes synchronously, without the need to add extra delay nodes. No broadcasting is left, but input data is needed throughout the graph.

An implementation of this graph could allocate each set of sequential nodes to a different PE. In such a case, the input matrix needs to be pre–loaded on the array before computation may start. The same scheme was derived through a different methodology in [20]. However, the efficiency of such approach is low, since different processors have widely varying number of operations to perform. For example, the top leftmost PE has only one operation to compute (i.e., only one node of the graph), while the lower rightmost PE has n operations.

From the utilization time for each node in Figure 17, we can see that in each row of the graph only one node is in use at every cycle. Therefore, it is possible to share one PE among the nodes of a row by combining all nodes in each row of the graph into a single one. This corresponds to collapsing paths in the graph where nodes have different utilization times. The resulting graph is shown in

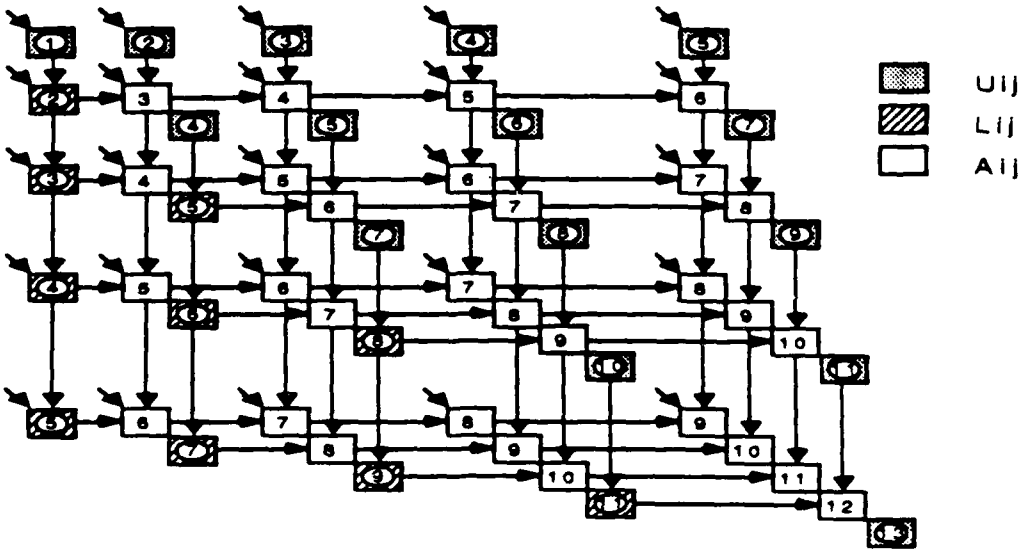

Figure 17: LU-decomposition graph without data broadcasting

Figure 18. A implementation of this graph leads to a triangular array for LU-decomposition, with utilization 0.732, twice that of other square arrays proposed for such computation [19],[20].

5 Conclusions

We have presented a research proposal for the development of a systematic methodology for the design of arrays of PEs for matrix algorithms. The proposed methodology uses a fully-parallel dependence graph of the algorithm as the description tool, because such notation exhibits the optimal features of the algorithm regarding delay, utilization of operation units, and throughput. The methodology consists of the systematic application of transformations on the graph, to fulfill restrictions which render the fully-parallel graph inadequate as an implementation.

We have described the features of a few basic transformations. Their application has allowed us to show that different transformations on a graph may lead to entirely different computing structures for a given algorithm. The selection of an adequate transformation is thus directed by the specific restrictions imposed on the implementation of the algorithm. Those restrictions include issues such as I/O bandwidth, regularity in the interconnection among processing elements (PEs), utilization of those PEs, limited data broadcasting, local communications, data synchronization.

We have presented the application of a preliminary version of this methodology to two known matrix algorithms, namely matrix multiplication and LU-decomposition. We have shown that the approach produces structures which correspond to proposed systolic arrays for these computations, as well as structures which exhibit better efficiency than those arrays. The methodology has been shown as capable to handle and relate features of the algorithm and the implementation, in a unified manner.

Although some of the transformations applied seem of general use, they are not an exhaustive

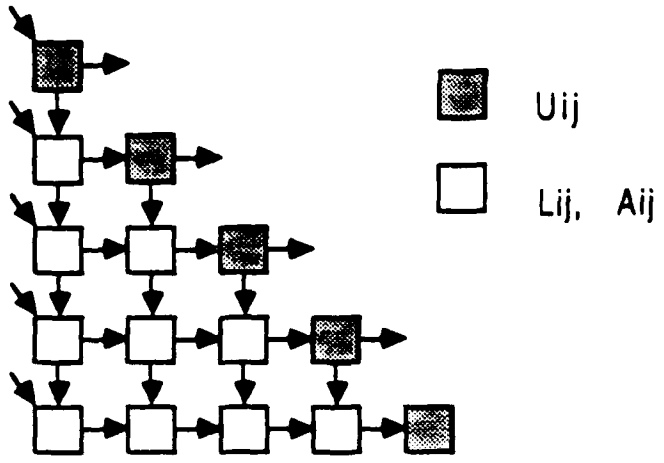

Figure 18: LU-decomposition graph with combined nodes

collection for all matrix computations. In addition, it seems that there are cases where transformations specific to a given algorithm are needed. The objectives of the proposed research include the identification and formal definition of a larger set of transformations, for a more varied class of matrix algorithms than what has been presented here. Additional transformations are also needed to include cases such as the computation of algorithms in arrays smaller than the dimensions of the matrix. A mechanism to define measures of efficiency in the design, and to evaluate implementations according to those measures is also needed. The ultimate goal of the proposed research is to provide the designer with a collection of transformations, which are systematically applied to a target algorithm. In this way, the design process becomes a search, in the space of solutions available through the transformations, for the alternative which offers the best cost-performance trade-offs.

References

[1] H. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[2] J. Fortes, K. Fu, and B. Wah, "Systematic approaches to the design of algorithmically specified systolic arrays," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 300–303, 1985.

[3] J. Moreno and T. Lang, "Replication and pipelining in multiple instance algorithms," in *International Conference on Parallel Processing*, pp. 285–292, 1986.

[4] J. Moreno and T. Lang, "A multilevel pipelined processor for the Singular Value Decomposition," in *SPIE Real-Time Signal Processing IX*, 1986.

[5] J. Speiser and H. Whitehouse, "Parallel processing algorithms and architectures for real-time signal processing," in *SPIE Real-Time Signal Processing IV*, pp. 2–9, 1981.

[6] J. Ullman, *Computational Aspects of VLSI*, ch. 2. Computer Science Press, 1984.

[7] R. Brent and H. Kung, "Area-time complexity of binary multiplier," *Journal of ACM*, vol. 28, no. 3, pp. 521–534, 1981.

[8] J. Symanski, "Implementation of matrix operations on the two-dimensional systolic array testbed," in *SPIE Real-Time Signal Processing VI*, pp. 136–142, 1983.

[9] J. Blackmer, G. Frank, and P. Kuekes, "A 200 million operations per second (MOPS) systolic processor," in *SPIE Real-Time Signal Processing IV*, pp. 10–18, 1981.

[10] A. Fisher, H. Kung, and L. Monier, "Architecture of the PSC: a programmable systolic chip," in *10th Annual Symposium on Computer Architecture*, pp. 48–53, 1983.

[11] G. Li and B. Wah, "The design of optimal systolic arrays," *IEEE Trans. Computers*, vol. C-34, pp. 66–77, Oct. 1984.

[12] I. Ramakrishnan and P. Varman, "An optimal family of matrix multiplication algorithms on linear arrays," in *International Conference on Parallel Processing*, pp. 376–383, 1985.

[13] M. Sheeran, "μFP – an algebraic VLSI design language," Technical Monograph PRG-39, Oxford University Computing Laboratory, University of Oxford, Sep. 1984.

[14] C. Guerra and R. Melhem, "Synthesizing non-uniform systolic designs," in *International Conference on Parallel Processing*, pp. 765–771, 1986.

[15] H. Kung and W. Lin, "An algebra for systolic computation," in *Conference on Elliptic Problem Solvers*, pp. 141–160, 1983.

[16] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *11th Annual Symposium on Computer Architecture*, pp. 208–214, 1984.

[17] M. Chen, "Synthesizing VLSI architectures: dynamic programming solver," in *International Conference on Parallel Processing*, pp. 776–784, 1986.

[18] V. Weiser and A. Davis, "A wavefron notation tool for VLSI array design," in *VLSI Systems and Computations*, (H. Kung et al., ed.), pp. 226–234, Computer Science Press, Oct. 1981.

[19] H. Kung, "Let's design algorithms for VLSI systems," in *CALTECH Conference on VLSI*, pp. 65–90, 1979.

[20] D. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proceedings of the IEEE*, vol. 71, pp. 113–120, Jan. 1983.

[21] J. Moreno and T. Lang, "Design of special-purpose arrays for matrix computations," in *submitted to SPIE Real-Time Signal Processing X*, 1987.

[22] W. Miranker and A. Winkler, "Space–time representations of computational structures," *Computing*, vol. 32, pp. 93–114, 1984.

[23] I. Ramakrishnan, D. Fussell, and A. Silberschatz, "On mapping homogeneous graphs on a linear array-processor model," in *International Conference on Parallel Processing*, pp. 440–447, 1983.

[24] T. Barnwell and D. Schwartz, "Optimal implementation of flow graphs on synchronous multiprocessors," in *Asilomar Conference on Circuits and Systems*, pp. 188–193, 1983.

[25] J. Jover and T. Kailath, "Design framework for systolic-type arrays," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 8.5.1–8.5.4, 1984.

[26] M. Lam and J. Mostow, "A transformational model of VLSI systolic design," *IEEE Computer*, pp. 42–52, Feb. 1985.

[27] R. Chapman, T. Durrani, and T. Willey, "Design strategies for implementing systolic and wavefront arrays using OCCAM," in *International Conference on Acoustics, Speech and Signal Processing*, pp. 292–295, 1985.

[28] K. Hwang and Y. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems," *IEEE Trans. Computers*, vol. C–31, pp. 1215–1224, Dec. 1982.

[29] J. Navarro, J. LLaberia, and M. Valero, "Solving matrix problems with no size restriction on a systolic array processor," in *International Conference on Parallel Processing*, pp. 676–683, 1986.

[30] D. Moldovan, C. Wu, and J. Fortes, "Mapping an arbitrarily large QR algorithm into a fixed size VLSI array," in *International Conference on Parallel Processing*, pp. 365–373, 1984.

[31] H. Chuang and G. He, "Design of problem–size independent systolic array systems," in *International Conference on Computer Design*, pp. 152–157, 1984.

International Conf. on Systolic Arrays
May 25-27-1988 San Diego, CA

ON PARTITIONING THE FADDEEV ALGORITHM

JAIME H. MORENO, TOMAS LANG *
Computer Science Department
University of California Los Angeles
Los Angeles, Calif. 90024

ABSTRACT

We present the derivation of partitioned schemes for computing the Faddeev algorithm, using a graph-based methodology. Such implementations are obtained by performing transformations on the fully-parallel dependence graph of the algorithm. We derive linear and two-dimensional structures and evaluate them in terms of throughput, I/O bandwidth, utilization of PEs and overhead due to partitioning. We also compare our partitioned implementations with schemes previously proposed. We show that throughput of both linear and two-dimensional arrays derived here tends to $(3m)/(7n^3)$, where m is the number of cells, and utilization tends to 1. We derive a two-dimensional scheme that is more efficient and has less overhead than others previously proposed. Moreover, we show that, for partitioned implementations with the same number of cells, a linear array performs better, its implementation is easier and it is better suited for fault-tolerant capabilities than a two-dimensional one.

INTRODUCTION

The implementation of matrix algorithms as collections of regularly connected processing elements (arrays of PEs) has been extensively studied lately. In many cases it is required to compute several algorithms in the same structure and to process large or variable-size matrices using a small array. One approach towards solving the first issue consists of using a general-purpose algorithm within a class of problems. Such is the case of the Faddeev algorithm [1], which is capable of performing a variety of matrix computations. This generality involves some overhead and cost, which in this case consists of performing additional computations. The second issue mentioned above is usually solved by decomposing the execution of the algorithm into sub-algorithms so that sub-algorithms fit into a target array. This is known as *partitioning* [2,3].

Several arrays to compute the Faddeev algorithm have been discussed in the literature. Nash and Hansen [4] have proposed a trapezoidal array for fixed-size problems and an implementation of their scheme has been presented in [5]. The same structure is used in [6] to partition the algorithm to fit into small arrays. The Faddeev algorithm is also used in [7] for fixed-size and variable-size problems, and in [8] for partitioned implementation in a two-dimensional array of transputers.

The implementations listed above exhibit low utilization of cells, and/or significant overhead due to partitioning. In addition, they have not been derived using a methodology. In this paper, we present the application to the Faddeev algorithm of a technique to partition the execution of algorithms in arrays of PEs. This is a transformational approach, which uses a fully-parallel dependence graph as the description of the algorithm. The graphical nature of this approach makes it easier to use than other design techniques. We briefly present our method and use it to devise arrays which implement the Faddeev algorithm in partitioned mode. (A complete description of the method and its applications are given in [9,10].) We evaluate the arrays obtained in terms of throughput, I/O bandwidth, utilization of PEs and overhead due to partitioning. We compare the

*J. Moreno has been supported by an IBM Computer Sciences Fellowship. This research has also been supported in part by the Office of Naval Research, Contract N00014-83-K-0493 "Specifications and Design Methodologies for High-Speed Fault-Tolerant Algorithms and Structures for VLSI"

arrays derived here with those previously proposed and show that our structures are more efficient and have less overhead. Moreover, we show that for partitioned implementations with the same number of cells, a linear array has better characteristics than a two–dimensional one.

FADDEEV ALGORITHM

The Faddeev algorithm [1] evaluates the expression $CX + D$ subject to the condition $AX = B$, where A, B, C, D are given matrices and X is a column vector. The algorithm can be expressed by representing input data as the extended matrix $\begin{pmatrix} A & B \\ -C & D \end{pmatrix}$ and performing linear combinations on this extended matrix, with the objective of transforming C into a matrix of zeroes.

Representing the operations performed as $(-C + WA)$ and $(D + WB)$, the annulment of C requires that $W = CA^{-1}$. Consequently, $D + WB = D + CA^{-1}B$. Since $X = A^{-1}B$, the expression $D + WB$ becomes $D + CX$, which is the desired result.

We use the modified Faddeev algorithm proposed by Nash and Hansen [4], in which an orthogonal factorization capability is added for numerical stability and to allow the coefficient matrix to be non–square. Such algorithm is a two–step process: first it triangularizes matrix A by Givens' rotations and also applies such rotations to matrix B, then it performs Gaussian elimination on C using the diagonal elements of the rotated matrix A as pivots and applies the same transformation to D.

The dependence graph of the modified Faddeev algorithm for 4 by 4 matrices is shown in Figure 1, after replacing data broadcasting by pipelining [11,12]. Delay nodes have been added to enhance communications regularity between nodes of the graph and to obtain nodes with at most one external input. Operation nodes correspond to computing multiply/add, division, rotation angle and rotations. For simplicity, we assume that all these nodes have the same computation time. The validity of such an assumption is highly implementation–dependent, as suggested by studies about the design of special–purpose cells [13,14].

We can distinguish four sections in Figure 1, namely those used to operate on the four different matrices composing the algorithm. In the top–left section, diagonal elements of matrix A are used to compute rotation angles. Such angles are broadcasted horizontally to the remaining elements of A on the same row and to elements of B also on the same row. All these elements are rotated according to such angles. Elements of the resulting triangular matrix Q and the rotated matrix B' flow towards the lower sections of the graph. In the lower–left section, diagonal elements of Q are used as pivots to perform Gaussian elimination on matrix C. Pivots are broadcasted horizontally and used together with elements of B' to perform the same transformation on matrix D.

In the next section, we use the dependence graph described above to discuss the design of arrays for partitioned execution of the Faddeev algorithm.

PARTITIONING THE FADDEEV ALGORITHM

For large size matrices, a dependence graph as the one shown in Figure 1 has too many nodes and the communication requirements and I/O bandwidth are complex and expensive. As a consequence, a pipelined implementation [11] of such graph is not feasible and the algorithm must be partitioned for execution in a small array. We briefly describe here our approach towards partitioning and present its application to the Faddeev algorithm. We refer the reader to [9] for further details on the methodology used.

Partitioning procedure. Our approach to partitioning consists of applying the following three–step procedure:

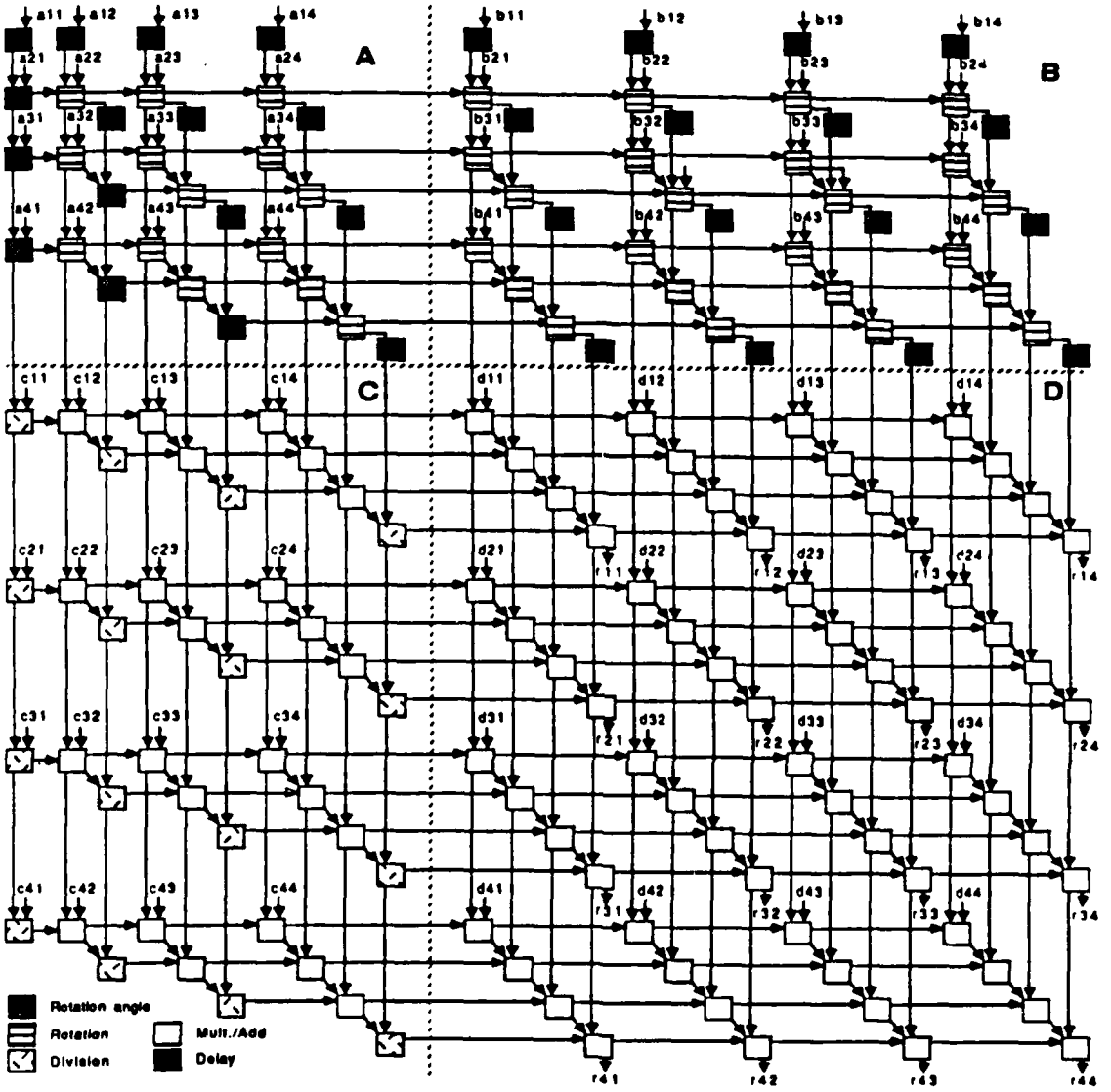

Figure 1: Fully-parallel dependence graph for Faddeev algorithm with no broadcasting

1. Transform the fully-parallel dependence graph to remove properties undesirable for an implementation, such as data broadcasting or bi-directional data flow. Procedures for these purposes have been proposed in [11,12].

2. Transform the graph obtained in (1) into a new graph, which we call the *G-graph*, by collapsing groups of nodes into new nodes (*G-nodes*). The objective of this transformation is to obtain a graph more suitable for partitioning, that is, with simple communication requirements. An example is shown in Figure 2, where sets of consecutive nodes in diagonal paths have been collapsed into *G-nodes*. Consequently, the number of nodes in the G-graph is smaller than the number of nodes in the graph used as input to this transformation.

3. Map G-nodes to a target array with m cells by scheduling sets of m neighbor G-nodes (a *G-set*) for concurrent computation, as shown in Figure 3. G-sets scheduled successively are executed in overlapped (pipelined) manner in the array. The selection of G-sets depends on the structure of the target array. In addition, for maximum utilization, all nodes in a G-set should have the same computation time.

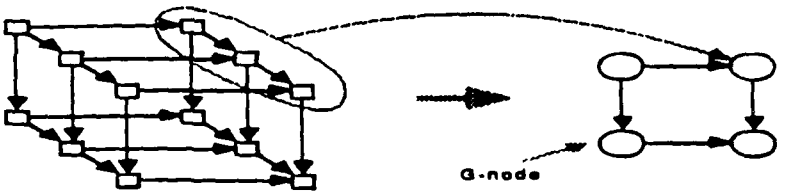

Figure 2: Collapsing primitive nodes into G–nodes

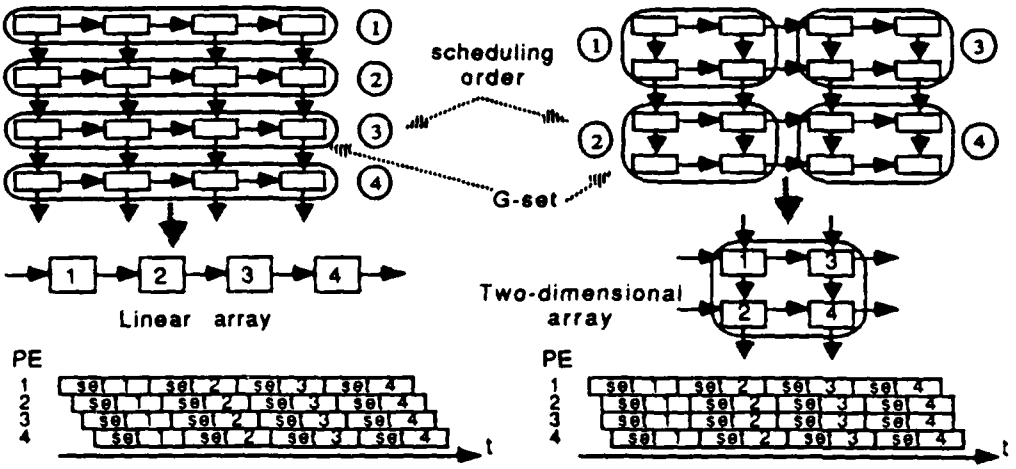

Figure 3: Mapping G–graph into linear and two–dimensional arrays

Partitioning the Faddeev algorithm for linear arrays

We apply now the procedure outlined above to partition the execution of the Faddeev algorithm for n by n matrices so that it fits in a linear structure with m cells, where $m \ll n$. Since the graph in Figure 1 doesn't have undesirable properties for implementation, we do not need to perform step 1 in the procedure above. To obtain the G–graph as indicated in step 2, we consider here the case of collapsing each vertical path of the graph in Figure 1 into a G–node (collapsing horizontal paths is also an alternative). Grouping vertical paths leads to the trapezoidal graph shown in Figure 4. In this graph, G–nodes of an horizontal path have the same computation time but such time decreases for lower horizontal paths of the G–graph.

In the last step of our procedure, we map G–sets from the transformed graph onto a linear array by selecting G–sets of m G–nodes in horizontal paths, as shown in Figure 5a. Scheduling of such G–sets is discussed later. Intermediate results from G–sets are saved in external memories. These intermediate results include rotation angles and pivots flowing horizontally, and rotated and pivoted rows flowing vertically. Such data is available at the boundary of the set, so that saving it in external memories is straight–forward. The structure resulting from this approach is shown in Figure 5b. This array enjoys maximal utilization because all G–nodes executed concurrently have the same computation time, except when executing boundary sets in some horizontal paths which might not use all cells in the array. The number of connections to external memories is $m + 1$.

Partitioning the Faddeev algorithm for two–dimensional arrays

We apply now our partitioning procedure to obtain two–dimensional arrays for the Faddeev algorithm. The first two steps of such procedure are the same as for linear arrays, so that we use the G–graph obtained above which is shown in Figure 4.

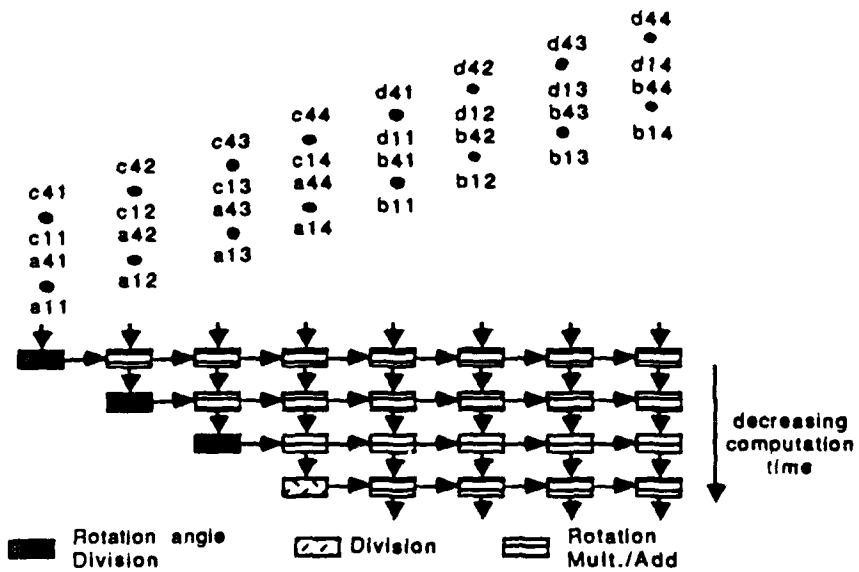

Figure 4: Trapezoidal graph from grouping vertical paths

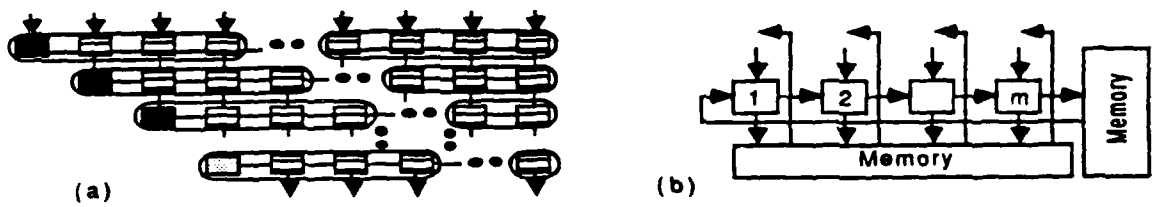

Figure 5: Partitioned linear array for the Faddeev algorithm

Mapping the trapezoidal G-graph for execution in a two-dimensional structure with m cells requires to simulate a triangular array and a square array, because those are the major components of the G-graph. Both requirements can be fulfilled in a square array, with the proper control signals. G-sets are selected as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes, excepting the leftmost sets which are composed of triangular blocks of G-nodes, as shown in Figure 6a. As in the linear case, intermediate results are saved in external memories. The structure resulting from this approach is shown in Figure 6b. Utilization of this array is not maximal, because the computation time of G-nodes is not the same for all nodes in a G-set. The number of connections to external memories is $2\sqrt{m}$.

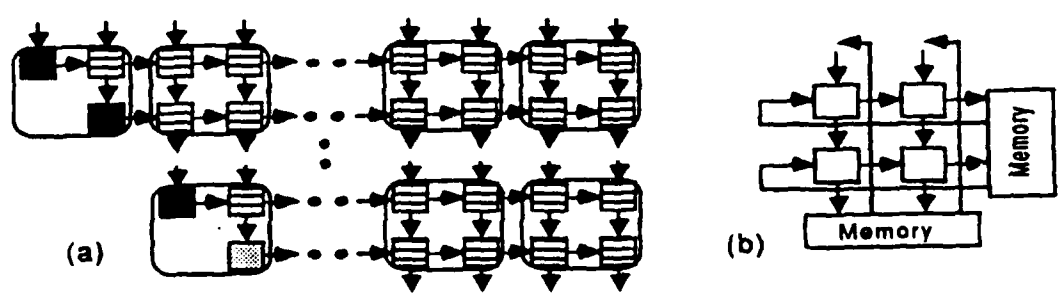

Figure 6: Two-dimensional partitioning of the Faddeev algorithm

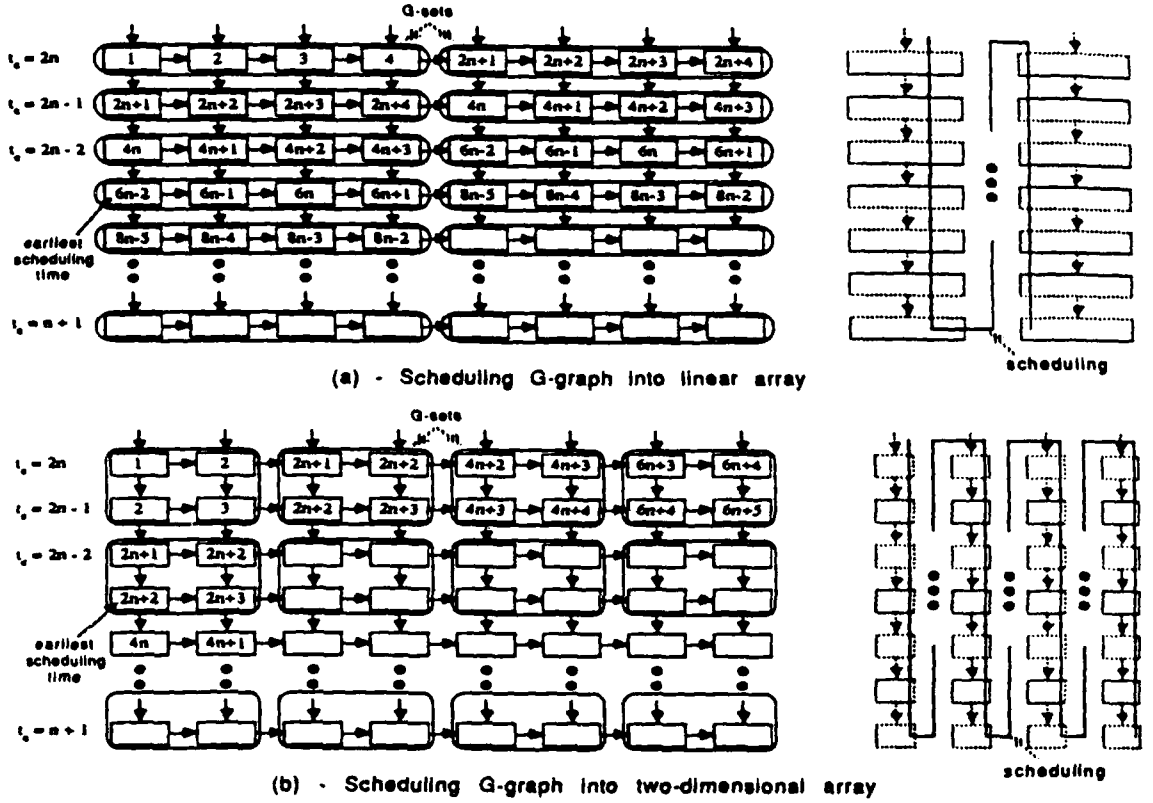

Figure 7: Scheduling G–graph into linear and two–dimensional arrays

Scheduling and I/O bandwidth in partitioned Faddeev algorithm

We discuss now the scheduling of G–sets mapped onto linear and two–dimensional arrays. To illustrate such scheduling, we use the G–graph shown in Figure 7 (this graph can be regarded as the internal portion of a large–size G–graph). Nodes in Figure 7 have been tagged with their earliest scheduling time relative to a reference time t_i. For each horizontal path of this graph, t_c is the computation time of G–nodes in such path.

Scheduling of G–sets must take into account the dependences among sets. Because of the pipelined nature of data flow within the array, a G–set can't be scheduled for computation until the required data produced by predecessor G–sets is available. However, computation time of nodes in a G–set is $O(n)$ while the length of dependences through the array is $O(m)$ because there are only m cells. Since $m \ll n$, data needed to schedule execution of the next G–set is available before the G–set in execution completes. Consequently, scheduling needs to consider only the dependences between G–sets.

Mapping onto a linear array was performed by composing a G–set with nodes in horizontal paths, because such nodes have the same computation time. Scheduling of G–sets can be done by horizontal (or vertical) paths, that is, by scheduling all G–sets in an horizontal (or vertical) path before scheduling G–sets on another path. For I/O bandwidth reasons discussed below, we choose to schedule G–sets by vertical paths as depicted in Figure 7a. This figure shows that G–sets can be scheduled in pipelined mode in a simple manner. Scheduling G–sets for execution in a two–dimensional array is similar to the linear array, as illustrated in Figure 7b.

A host feeding input data to the array needs to provide only the elements appearing as input to nodes at the topmost horizontal path of the G–graph. Intermediate values are saved in and obtained

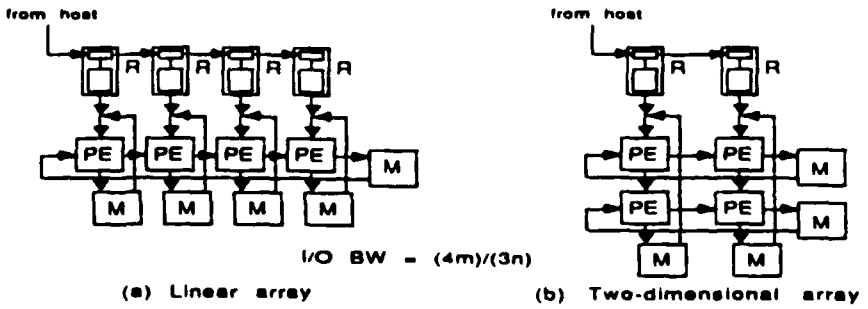

Figure 8: I/O bandwidth in partitioning Faddeev algorithm

from external memories attached to the array. To reduce the rate at which data has to be provided to the array, nodes at the top of the G-graph should not be scheduled consecutively. In such a case, the host needs to feed data to the array at a rate lower than one input per cell per cycle and utilization of I/O connections can be increased by decoupling computation from data transfer, as proposed in [11]. Such approach leads to the I/O structure shown in Figure 8, where the host feeds data to the array through a chain of registers (the R blocks in the figure). Each block R consists of a register and memory. Data from the host flows in pipelined mode through the registers and is stored in the memories. When a G-set from the top of the graph is scheduled for execution, data is read from the memories into the PEs while new data is transferred from the host.

Since each node at the top of the G-graph receives $2n$ data elements from the host, I/O bandwidth is given by

$$D_{I/O} = \frac{2nm}{\sum_{k=1}^{n} t_{c_k}} = \frac{2nm}{(2n+1)n - \frac{1}{2}n(n+1)} \approx \frac{4}{3}\frac{m}{n} \quad \text{[words/cycle]}$$

where t_{c_k} is the computation time of G-nodes in the k-th row of the G-graph. Under the conditions described above, *linear and two-dimensional arrays have the same I/O bandwidth from the host.*

EVALUATING ARRAYS FOR PARTITIONED FADDEEV ALGORITHM

We evaluate now the characteristics of the arrays presented in the previous section. Such evaluation is based on information obtained from the dependence graphs of the algorithm, both the original and the transformed graphs. We also compare those arrays with some schemes previously proposed. We use throughput, input bandwidth, utilization, and overhead due to partitioning as performance measures. We assume the same number of cells for the different arrays.

Utilization of the arrays is computed as number of nodes in the dependence graph divided by m/T, where m is number of cells and T is throughput. We ignore delay nodes shown in Figure 1 because those nodes don't perform useful computation. The number of nodes N is given by

$$N = \sum_{i=0}^{n-1}(2n-i)(2n-i-1) = \frac{7}{3}n^3 - \frac{n}{3}$$

The expression above is different from the one given by De Groot et al. in [8], because they count as operations cycles when a cell is waiting to collect the first two operands before performing the first operation in a group (i.e., delay nodes due to the single-input capacity of cells). Consequently, their measure of complexity of the algorithm (i.e., $3n^3 + n^2$ operations) is greater than the actual value.

Throughput is determined by the computation time of the busiest cell in the array. Such information is obtained from mapping the G-graph onto the target array. In the following subsections, we present the derivation of the corresponding expressions for the different arrays.

Linear array. When scheduling the G-graph shown in Figure 4 for execution in a linear array with m cells, the i-th horizontal path has length $2n - i + 1$ and it is mapped in $\lceil (2n - i + 1)/m \rceil$ sets (we assume that n/m is an integer). Each G-node in such path consists of $2n - i + 1$ operations. Therefore, the array is used for

$$\sum_{i=0}^{n-1} \left\lceil \frac{2n-i}{m} \right\rceil (2n-i) = \sum_{k=0}^{\frac{n}{m}-1} \left[\left(\frac{2n}{m} - k \right) \sum_{i=0}^{m-1} (2n - i - km) \right] \approx \frac{28n^3 - 9n^2(m-1)}{12m}$$

and throughput is

$$T_{linear} = \frac{12m}{28n^3 - 9n^2(m-1)} \quad [\text{eval}]^{-1}$$

Utilization is given by

$$U_{linear} = \frac{\sum \text{nodes}}{m/T} = \frac{28n^2 - 4}{28n^2 - 9n(m-1)}$$

Therefore, for large n, utilization tends to 1 and throughput tends to $\frac{3}{7}(m/n^3)$.

Square array. In the square array proposed here, G-sets are selected as square blocks of $\sqrt{m}$ by $\sqrt{m}$ nodes. Therefore, G-sets are mapped by strips of $\sqrt{m}$ by $\sqrt{m}$ blocks. The i-th horizontal strip has $(2n - (i-1)\sqrt{m})/\sqrt{m}$ G-sets. The computation time of these sets is given by the computation time of nodes in the first horizontal path of the set, which corresponds to $(2n - (i-1)\sqrt{m})$ operations. Therefore, the array is used for

$$\sum_{i=0}^{p-1} \left(\frac{2n - i\sqrt{m}}{\sqrt{m}} \right) (2n - i\sqrt{m}) = \frac{1}{\sqrt{m}} \sum_{i=0}^{p-1} (2n - i\sqrt{m})^2 \approx \frac{7}{3} \frac{n^3}{m} \quad [\text{ops}]$$

where $p = n/\sqrt{m}$ is the number of strips of $\sqrt{m}$ by $\sqrt{m}$ sets of G-nodes needed to cover the entire G-graph. Throughput is

$$T_{square} = \frac{3m}{7n^3} \quad [\text{eval}]^{-1}$$

and utilization is given by

$$U_{square} = \frac{\sum \text{nodes}}{m/T} = \frac{7n^3 - n}{7n^3}$$

Therefore, for large n, utilization tends to 1 and throughput tends to $\frac{3}{7}(m/n^3)$

Nash et al. array. Nash et al. [6] use a square array to map their model of the algorithm. Such model consist of a bi-trapezoidal graph with corresponding nodes interconnected [10]. They map each of the trapezoidal sub-graphs independently, so that they require certain overhead in unloading/loading and skewing/de-skewing data. The derivation of the corresponding performance measures is given in [10]. Throughput of their implementation is

$$T_{\text{Nash}} = \frac{6m}{14n^3 + 9n^2\sqrt{m} + nm + 6m(\text{OVHD})} \quad [\text{ops}]^{-1}$$

where OVHD is the overhead in data transfers (such overhead has not been reported quantitatively). Utilization is given by

$$U_{\text{Nash}} = \frac{\sum \text{nodes}}{m/T} = \frac{14n^2 - 2}{14n^3 + 9n^2\sqrt{m} + nm + 6m(\text{OVHD})}$$

Consequently, Nash et al. implementation has the same throughput as the square array proposed above if there was no overhead. In practice, such throughput and utilization of the array are lower. In addition, their scheme exhibits complexity in the control required to perform those data transfers

Table 1: Performance measures for partitioned implementations with m cells

Array	Throughput [1/ops]	I/O BW	Utilization	Overhead
Linear	$\frac{12m}{28n^3-9n^2(m-1)}$	$\frac{4m}{3n+1}$	$\frac{28n^2-4}{28n^2-9n(m-1)} \rightarrow 1$	none
Square	$\frac{3m}{7n^3}$	$\frac{4m}{3n+1}$	$\frac{7n^3-n}{7n^3} \rightarrow 1$	none
De Groot	$\frac{12m-\sqrt{24m+1}+1}{36n^3}$	—	$\frac{(7n^2-1)(12m-\sqrt{24m+1}+1)}{108n^2m}$ $\rightarrow 7/9$	$O(n^2)$ storage
Nash	$\frac{6m}{14n^3+9n^2\sqrt{m}+nm+6m(ovhd)}$	$\frac{4m}{3n+1}$ + ovhd	$\frac{14n^2-2}{14n^3+9n^2\sqrt{m}+nm+6m(ovhd)}$	loading, skewing

into and out of the array. I/O bandwidth of Nash et al. scheme is higher than the square array above, because of the loading/unloading of data.

De Groot et al. partitioned scheme. De Groot et al. [8] implementation is an hypercube with transputers as nodes. They partition the algorithm by applying a technique known as *coalescing* [2] and evaluate their scheme considering that data communication is ten times slower than performing a single operation in a cell. The derivation of performance measures for their scheme is given in [10]. Throughput of their implementation is

$$T_{DeGroot} = \frac{12m - \sqrt{24m+1}+1}{36n^3} \quad [\text{ops}]^{-1}$$

and utilization is given by

$$U_{DeGroot} = \frac{\sum \text{nodes}}{N/T} = \frac{84n^2m - 12m + (7n^2-1)(1-\sqrt{24m+1})}{108n^2m}$$

Therefore, for large n, utilization tends to 7/9 and throughput tends to $\frac{1}{3}(m/n^3)$.

The results above are summarized in Table 1. We have not included I/O bandwidth for De Groot et al. scheme, because they transfer all data before starting the computation. From this table we infer that, for large n, both our linear and square arrays tend to the same throughput ($\frac{3}{7}(m/n^3)$) and utilization tends to 1. In addition, both exhibit the same I/O bandwidth from the host. These linear and square arrays have better performance measures than the array proposed by De Groot et al. and do not exhibit the overhead required in the scheme proposed by Nash et al.

In addition to the performance measures described above, a linear array is more advantageous for the Faddeev algorithm than a two–dimensional one because:

- it is simpler to implement
- for a finite value of n it has slightly higher utilization than the two–dimensional structure
- it is better suited to incorporate fault–tolerant capabilities (i.e., it's easier to skip a faulty cell in a linear array than to reconfigure a two–dimensional structure)

Consequently, we conclude that *for partitioned execution of the Faddeev algorithm, a linear array offers better performance and implementation than a two–dimensional array.*

CONCLUSIONS

We have presented the application of a graph–based methodology to derive partitioned implementations for the Faddeev algorithm. We have obtained linear and two–dimensional arrays for such algorithm, and we have compared these structures to others previously proposed. We have shown that the two–dimensional array derived here is more efficient and has less overhead than those other schemes. Moreover, we have shown that linear and two–dimensional arrays exhibit the same I/O bandwidth from the host, and utilization and throughput of both structures tend to the same values. We have concluded that, since performance measures of both arrays are identical, a linear array is better than a two–dimensional one because it is simpler to implement and is more suitable to incorporate fault–tolerant capabilities.

References

[1] D. Faddeev and V. Faddeeva, *Computational Methods of Linear Algebra*, pp. 150–158. W.H. Freeman and Co., 1963.

[2] J. Navarro, J. Llaberia, and M. Valero, "Partitioning: an essential step in mapping algorithms into systolic array processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[3] D. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. C-35, pp. 1–12, Jan. 1986.

[4] J. Nash and S. Hansen, "Modified Faddeev algorithm for matrix manipulation," in *SPIE Real–Time Signal Processing VII*, pp. 39–46, 1984.

[5] J. Nash, K. Przytula, and S. Hansen, "Systolic/cellular processor for linear algebraic operations," in *VLSI Signal Processing II*, (J. N. S.Y. Kung, R. Owen, ed.), pp. 306–315, IEEE Press, 1986.

[6] J. Nash, S. Hansen, and K. Przytula, "Systolic partitioned and banded linear algebraic computations," in *SPIE Real–Time Signal Processing IX*, pp. 10–16, 1986.

[7] H. Chuang and G. He, "A versatile systolic array for matrix computations," in *12th Annual Symposium on Computer Architecture*, pp. 315–322, 1985.

[8] A. D. Groot, E. Johansson, and S. Parker, "Systolic array for efficient execution of the Faddeev algorithm," in *SPIE Real–Time Signal Processing X*, pp. 86–93, 1987.

[9] J. Moreno and T. Lang, "Graph–based partitioning of matrix algorithms for systolic arrays," Technical Report CSD-880015, Computer Science Department, University of California Los Angeles, March 1988.

[10] J. Moreno and T. Lang, "Designing arrays for the Faddeev algorithm," Technical Report CSD-880013, Computer Science Department, University of California Los Angeles, March 1988.

[11] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations: preliminary results," in *SPIE Real–Time Signal Processing X*, pp. 53–65, 1987.

[12] J. Moreno and T. Lang, "Reducing the number of cells in arrays for matrix computations," Technical Report CSD-880014, Computer Science Department, University of California Los Angeles, March 1988.

[13] M. Ercegovac and T. Lang, "On-line scheme for computing rotation factors," in *8th Symposium on Computer Arithmetic*, pp. 196–203, 1987.

[14] J. Cavallaro and F. Luk, "CORDIC arithmetic for an SVD processor," in *8th Symposium on Computer Arithmetic*, pp. 215–222, 1987.